



Universidad Complutense de Madrid
Facultad de informática

Sistemas Informáticos 09-10

<e-Tutor> GE

<e-Tutor> Graphical Editor

Autores:

Miguel Esteban Écija

Javier Hernández Rodríguez

Esther Peña Rubio

Director:

José Luis Sierra Rodríguez

AGRADECIMIENTOS

Tras muchas horas dedicadas y muchos quebraderos de cabeza, este es el resultado de nuestro trabajo y esfuerzo.

Queremos agradecer a José Luis la oportunidad que nos ha brindado con este proyecto, y en especial a nuestros familiares, seres queridos y amigos, por servirnos de apoyo en los momentos difíciles y saber disfrutar con nosotros de las alegrías y los buenos ratos.

Se autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria como el código, la documentación y/o el prototipo desarrollado.

Miguel Esteban

Javier Hernández

Esther Peña

RESUMEN

En este proyecto de Sistemas Informáticos construimos una herramienta sencilla, que consiste en un editor gráfico para <e-Tutor> y facilita la creación de tutoriales sin necesidad de tener conocimientos de lenguajes de programación.

El editor permite asociar una serie de figuras que, procesadas secuencialmente, generarán un tutorial socrático. Este tipo de tutoriales pueden ser utilizados en multitud de campos, no sólo el informático, siempre dentro de un marco educativo.

El desarrollo de <e-Tutor> GE se apoya en diferentes tecnologías Eclipse, que permiten la construcción de una herramienta gráfica a partir de un modelo de datos.

ABSTRACT

In This Project we have built a simple tool which consists on a graphical editor for <e-Tutor> that makes it easier to create tutorials without any knowledge in programming languages.

The editor permits authors to associate certain figures, that sequentially processed, will generate a socratic tutorial. This kind of tutorials can be used for various matters, not only computing, but always within an educational framework.

The development of <e-Tutor> GE is supported by some Eclipse technologies, which allow building a graphical tool from a data model.

Tabla de contenidos:

1.- Introducción	12
1.1.-Objetivos	12
1.2.- Estructura de la memoria	13
2.- Revisión de conceptos y de tecnologías.....	14
2.1.- Sistemas tutores	14
2.2.- Tecnologías Eclipse:	18
2.2.1.- Introducción al IDE de Eclipse	18
2.2.2.- Plug-ins y Extensiones.....	21
2.2.2.1.- Proyectos de plug-in en Eclipse	22
2.2.2.2.- Extensiones	22
2.2.3.- EMF (Eclipse Modeling Framework)	23
2.2.4.- GEF (Graphical Editing Framework)	24
2.2.5.- GMF (Graphical Modeling Framework)	28
3.- La herramienta de autor para <e-Tutor>:	30
3.1.-Notación gráfica:.....	30
3.2.-La herramienta:	33
4.- Desarrollo e implementación del proyecto.	43
4.1.- Metodología de desarrollo:	43
4.2.- Proceso de desarrollo del modelo:	44
4.3.-Metamodelo <e-Tutor>GE:	45
4.4.- Editor Gráfico <e-Tutor>GE:	53
4.4.1.- Creación de un proyecto GMF	54
4.4.2.- Creación o importación de un modelo	55
4.4.3.- Generación del código del modelo	56
4.4.4.- Definición gráfica del modelo.....	57
4.4.5.- Definición de la Paleta de Herramientas	64
4.4.6.- Generación del mapeo de los elementos	68
4.4.6.- Generación de código	80
4.4.7.- Ejecución de la herramienta de creación de diagramas	82
4.5.- Plugins <e-Tutor>GE.....	85
4.5.1.- Creación de un proyecto Plugin.....	85

4.5.2.- Creación del archivo xml y modificación del MANIFEST	89
4.5.3.- Implementación del código	92
4.6.- Reconstrucción de <e-Tutor> GE	104
4.7.- Control de versiones de <e-Tutor> GE.....	106
4.7.1.- Versión básica <e-Tutor> GE	106
4.7.2.- Modificación de figuras	107
4.7.3.- Incorporación de la clase Feedback	107
4.7.4.- Compartments y tutoriales anidados	108
4.7.5.- Speech con archivo y editor de texto.....	108
4.7.6.- Plug-in para cargar archivos.....	109
4.7.7.- Plug-in para editar archivos.....	110
4.7.8.- Modificación del metamodelo para guardar los archivos	110
4.7.9.- Plug-in de ejecución y modelo definitivo	110
5.- Conclusiones y trabajo futuro	112
5.1.- Conclusiones	112
5.2.- Trabajo futuro	113
6.- Referencias bibliográficas	115
7.- Webgrafía	116

1.- Introducción

El problema abordado en este proyecto de Sistemas Informáticos es el de la creación de un editor gráfico para la herramienta <e-Tutor>. <e-Tutor> [13][14][12][10] es una herramienta experimental para el desarrollo de tutoriales creada en el grupo de Ingeniería del Software aplicada al e-Learning (grupo e-UCM) de la Facultad de Informática de la Universidad Complutense de Madrid. Anteriormente a este desarrollo, la única forma de crear tutoriales en <e-Tutor> era mediante el procesamiento de documentos xml. La limitación de este método para generar tutoriales era que, al ser xml un lenguaje informático, esta herramienta resultaba difícil de utilizar para personas que carecen de conocimientos de programación.

Lo que se pretende con este proyecto es facilitar la creación de tutoriales, de forma sencilla, mediante un editor gráfico compuesto por las diferentes figuras que constituyen un tutorial. Con esto se consigue que personas sin conocimientos del lenguaje xml, puedan desarrollar sus propios tutoriales.

1.1.-Objetivos

El principal objetivo de este proyecto es desarrollar un editor gráfico que permita la creación de tutoriales.

Inicialmente, se pretende ofrecer al usuario una interfaz sencilla basada en el uso de figuras de diferentes tamaños, formas y colores. El sistema deberá contener una paleta con todos los elementos que forman un tutorial y una ventana de edición donde se puedan añadir. Esta paleta deberá contener figuras que representen elementos

concretos tales como textos, preguntas, respuestas y las conexiones que permiten la unión de forma ordenada entre ellas.

La herramienta ofrecerá la posibilidad de personalizar el tutorial, asignándole por ejemplo colores de fondo, título u otras opciones gráficas.

Para facilitar la introducción de texto en el tutorial, el sistema ofrecerá la opción de cargar archivos ya escritos o editarlos dentro la propia herramienta.

Además, este editor tendrá la funcionalidad de poder ejecutar los tutoriales creados y generarlos automáticamente.

1.2.- Estructura de la memoria

Esta memoria está estructurada en secciones que abordan los diferentes aspectos del proyecto.

- *Sección 2:* ofrece una introducción a los sistemas tutores, y a las principales tecnologías de Eclipse utilizadas en el desarrollo de la aplicación.
- *Sección 3:* explica detalladamente el aspecto gráfico de la herramienta, las figuras que la componen y el proceso de creación de un tutorial, basado en una sucesión ordenada de los elementos.
- *Sección 4:* detalla tanto la metodología de trabajo empleada como la arquitectura e implementación del proyecto. En esta sección se profundiza sobre cada uno de los procesos requeridos para llevar a cabo el editor.
- *Sección 5:* aquí se exponen las distintas conclusiones obtenidas en la realización del proyecto, así como una serie de ideas para llevar a cabo en un posible trabajo futuro.

2.- Revisión de conceptos y de tecnologías

2.1.- Sistemas tutores

Un sistema tutor es un software que pretende emular la acción de un maestro, impartiendo un conocimiento nuevo mediante el diálogo interactivo con el alumno [15]. En él se sigue una estrategia de instrucción previamente definida por su creador, la cual irá guiando al estudiante hasta alcanzar una meta.

En nuestro caso vamos a centrarnos en los sistemas tutores socráticos [1][6], es decir, aquellos que conducen al alumno mediante una serie de preguntas y respuestas y le ayudan a aprender un concepto. Podríamos considerar esto como un diálogo “maestro-discípulo”, de ahí el término socrático. Consideramos que éste es un buen método de enseñanza ya que el buen profesor no es aquél que nos da la respuesta correcta, sino aquél que nos ayuda a encontrarla por nosotros mismos [9].

Cada vez con más frecuencia estamos viendo este tipo de tutores virtuales, que pueden desempeñar su función independientemente del lugar donde se encuentre el alumno, lo que es una gran ventaja.

Otra ventaja que podemos destacar es que los “profesores” no necesitan conocer ningún lenguaje de programación para crear este tipo de tutoriales, además de ser un método de enseñanza relativamente barato. Es por ello que muchos profesores pueden beneficiarse de este tipo de sistemas tutores para desarrollar, de forma sencilla, sus propias lecciones. El problema de todo esto es que al crear sistemas fáciles de aprender, estos deben ser bastante simples, lo que provoca que las lecciones sean de peor calidad. Pero

como ya hemos dicho, crear un sistema tutor más complejo y con más posibilidades requiere en algunos casos que el profesor tenga una base de programación que muchas veces no se da, lo que limita el uso de este tipo de tutoriales.

Como en el desarrollo de cualquier sistema, en la enseñanza asistida por ordenador (Computer Aided Learning - CAL -), se utilizan una serie de herramientas para el desarrollo del software, tales como: herramientas de especificación, herramientas de implementación, herramientas de construcción de prototipos y herramientas de mantenimiento [6]. En la Figura 2.1 se muestran las fases para el desarrollo de una herramienta CAL, según [6]. De acuerdo con este modelo de desarrollo:

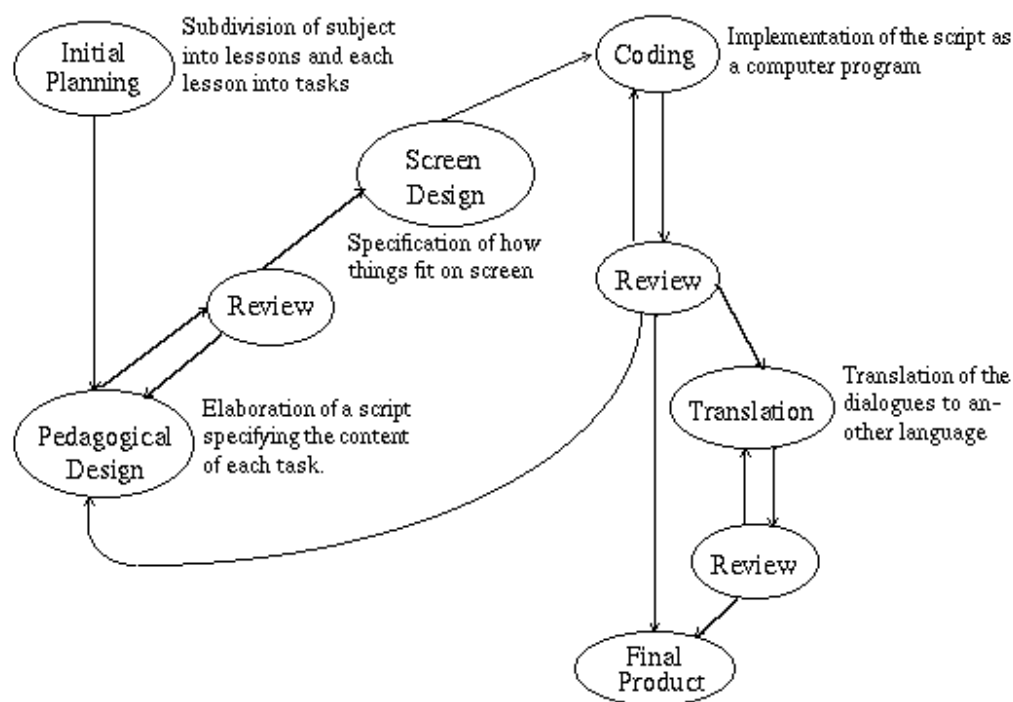


Figura 2.1 Fases para el desarrollo de una herramienta CAL

- Primeramente, haremos una división de la materia en lecciones y de cada lección en tareas, para de esta forma elaborar una especie de “plan inicial”.

-
- Una vez que tenemos la materia dividida en lecciones, elaboramos un guión que especifique el contenido de cada una de ellas, lo que llamaremos “diseño pedagógico”. En este punto, se pueden realizar revisiones por si fuera necesario cambiar la división de la materia del curso por otra más apropiada.
 - Lo siguiente será pensar cómo aparecerán las lecciones en la pantalla del ordenador, es decir, establecer un criterio para el diseño gráfico del tutorial.
 - A continuación, hay que implementar el código para que pueda ejecutarse como un programa. Una vez que tenemos el código y el tutorial ejecutando, es posible que queramos cambiar algo, por lo que sometemos al programa a una nueva revisión y, si fuera necesario, cambiaríamos el llamado “diseño pedagógico”.
 - Si finalmente el tutorial está adecuadamente terminado, podemos sugerir su traducción a otros idiomas para facilitar su distribución, con las consecuentes revisiones que esto conlleva.

Tras aplicar este modelo, podemos concluir que hemos elaborado una herramienta de enseñanza asistida por ordenador siguiendo unas pautas de desarrollo concretas.

En este proyecto hemos seguido el modelo <e-Tutor>, que está basado en los trabajos pioneros desarrollados por el equipo del Prof. Alfred Bork durante la década de los 80 [1][6]. Este modelo interpreta descripciones de tutoriales en los que el estudiante se somete a problemas, para los cuales se proponen una serie de soluciones a través del ya mencionado diálogo “socrático”.

Mediante estas soluciones el tutorial decidirá el próximo paso a llevar a cabo. Cada uno de estos caminos llevará a un evento diferente, en función de la respuesta y del número de veces que una respuesta concreta ha sido dada por el alumno.

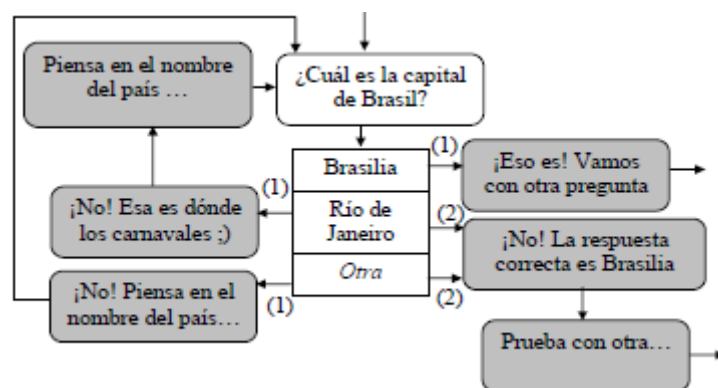


Figura 2.2 Ejemplo de tutorial socrático

En la Figura 2.2 tenemos un ejemplo de tutorial socrático, en el que se puede apreciar el diálogo entre el profesor y el alumno. En función de la respuesta de éste último, se da un consejo u otro para ayudarle a contestar correctamente.

En la Figura 2.3 se muestra un modelo a alto nivel de la estructura de los tutoriales en <e-Tutor>.

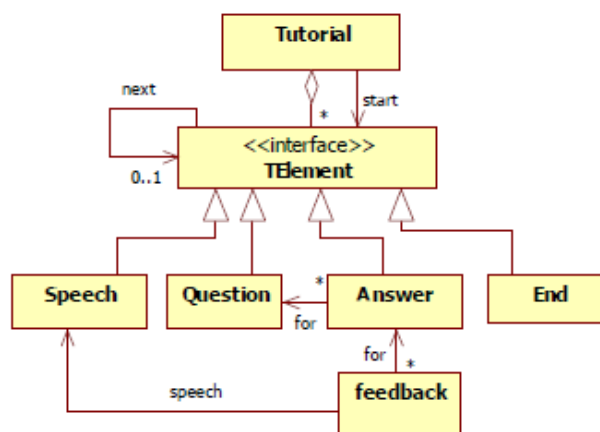


Figura 2.3 Modelo alto nivel <e-Tutor>

Podemos observar que la clase tutorial contiene elementos, los cuales pueden ser Speech, Question, Answer o End. Un Answer siempre va asociado a un Question. También existe otra clase, feedback, que relaciona un Answer con un Speech, y se refiere al número de veces

que una respuesta ha sido elegida por el alumno. En función del número de respuesta, se mostrará un mensaje u otro. Todo tutorial comienza con un elemento, y puede tener siguiente o no.

2.2.- Tecnologías Eclipse:

2.2.1.- Introducción al IDE de Eclipse

Para el desarrollo de la herramienta de autor se eligió el entorno de trabajo Eclipse [4]. Eclipse es un entorno de desarrollo integrado de código abierto multiplataforma para el desarrollo de aplicaciones. La plataforma se usa típicamente para el desarrollo en IDE (Entornos de desarrollo integrados), como puede ser el IDE de Java (JDT). El trabajo en Eclipse consiste en un proyecto central que incluye un framework genérico para la integración de herramientas, y un entorno de desarrollo Java construido usando el framework anterior.

Eclipse es una plataforma de cliente enriquecido, que ofrece multitud de opciones para el usuario:

- *Plataforma principal*: inicio de Eclipse, ejecución de plugins
- *OSGi*: una plataforma que permite la instalación y manejo de componentes y aplicaciones Java de manera remota.
- *El Standard Widget Toolkit (SWT)*: un widget toolkit portable.
- *JFace*: manejo de archivos, manejo de texto, editores de texto.
- *El Workbench de Eclipse*: vistas, editores, perspectivas, asistentes...

Otros proyectos extienden el framework, núcleo para soportar tipos de herramientas y entornos de desarrollo específicos, entre los que encontramos PDE (Plug-in Development Environment) [4][2], EMF

(Eclipse Modeling Framework) [16][8], GEF (Graphical Editing Framework) [2][5][8] y GMF (Graphical Modeling Framework) [5].

El IDE de Eclipse está basado en plug-ins, proporcionando así toda la funcionalidad que el cliente necesita, a diferencia de otros entornos de trabajo de estructura monolítica, donde aparecen todas las funcionalidades incluidas, las necesite o no el usuario. Así podemos extender Eclipse para otros lenguajes de programación o incluso para la gestión de bases de datos. Por otro lado, Eclipse permite al usuario crear sus propios plug-ins con el fin de personalizar distintas funciones del IDE. Estos plug-ins se definen como puntos de extensión, los cuales se pueden diseñar e incluir en otros proyectos a gusto del usuario. En el desarrollo de la herramienta hacemos uso de esta característica, para añadir diversas funcionalidades al editor, que serán estudiadas con posterioridad.

Eclipse está compuesto de tres subproyectos principales: la Plataforma Eclipse, Java Development Tool y el Plug-in Development Environment. El éxito de la Plataforma Eclipse depende de cómo sea capaz de admitir una amplia gama de herramientas de desarrollo para reproducir lo mejor posible las herramientas existentes en la actualidad.

Eclipse lo forman el núcleo, el entorno de trabajo (Workspace), el área de desarrollo (Workbench), la ayuda al equipo (Team support) y la ayuda o documentación (Help):

- *Núcleo*: su tarea es determinar cuáles son los plug-ins disponibles en el directorio de plug-ins de Eclipse. Cada plug-in tiene un fichero XML manifest que lista los elementos que necesita de otros plug-ins así como los puntos de extensión que ofrece. Como la cantidad de plug-ins puede ser muy grande, sólo se cargan los necesarios en el momento de ser utilizados

con el objeto de minimizar el tiempo de arranque de Eclipse y el tiempo de carga de recursos.

- *Entorno de trabajo*: maneja los recursos del usuario, organizados en uno o más proyectos. Cada proyecto corresponde a un directorio en el directorio de trabajo de Eclipse, y contiene archivos y carpetas.
- *Interfaz de usuario*: muestra los menús y herramientas, y se organiza en perspectivas que configuran los editores de código y las vistas. A diferencia de muchas aplicaciones escritas en Java, Eclipse tiene el aspecto y se comporta como una aplicación nativa. No está programada en Swing, sino en SWT (Standard Widget Toolkit) y JFace (juego de herramientas construida sobre SWT), que emula los gráficos nativos de cada sistema operativo [4][2]. Este ha sido un aspecto discutido sobre Eclipse, porque SWT debe ser portada a cada sistema operativo para interactuar con el sistema gráfico. En los proyectos de Java puede usarse AWT y Swing salvo cuando se desarrolle un plug-in para Eclipse.
- *Ayuda al grupo*: este plug-in facilita el uso de un sistema de control de versiones para manejar los recursos en un proyecto del usuario, y define el proceso necesario para guardar y recuperar de un repositorio. Además, Eclipse incluye un cliente para CVS [17].
- *Documentación*: al igual que el propio Eclipse, el componente de ayuda es un sistema de documentación extensible. Los proveedores de herramientas pueden añadir documentación en formato HTML y, usando XML, definir una estructura de navegación.

Por otro lado, Eclipse provee al programador con una serie de frameworks muy ricos para el desarrollo de multitud de aplicaciones, como pueden ser editores gráficos, manipulación de modelos, aplicaciones web, etc. Un framework es una estructura conceptual y tecnológica de soporte definida, normalmente con artefactos o módulos de software concretos, con base en la cual otro proyecto de software puede ser organizado y desarrollado [3]. En el proyecto que nos ocupa, nos hemos servido de tres de los frameworks de Eclipse: EMF (Eclipse Modeling Framework), GEF (Graphical Editing Framework) y GMF (Graphical Modeling Framework).

2.2.2.- Plug-ins y Extensiones

El desarrollo de la herramienta está completamente basado en el uso de plug-ins. Un plug-in es la mínima unidad de la plataforma que puede ser desarrollado por separado y que la aporta una nueva funcionalidad. Pueden ser freeware y de pago; incluso el programador puede programar los suyos propios, utilizando los proyectos de plug-in. Se instalan descomprimiendo el zip del plug-in en el directorio plugins de Eclipse. Los proyectos de plug-in tienen la particularidad de poder ser extendidos mediante cualquier otro plug-in, así como servir de extensión a cualquier otro proyecto. En cuanto a las extensiones, nos proporcionan una manera de añadir ciertas funcionalidades a nuestros proyectos, que pueden ser crear una nueva vista de Eclipse, añadir nuevos menús que implementen diversas acciones, crear editores, y en definitiva, infinidad de otras opciones. A continuación, ofrecemos una breve explicación de los proyectos de plug-in de Eclipse y de los puntos de extensión.

2.2.2.1.- Proyectos de plug-in en Eclipse

A la hora de hablar de un proyecto de plug-in debemos destacar dos componentes: el documento MANIFEST.MF y el archivo plugin.xml:

En MANIFEST.MF quedaran reflejadas las dependencias de nuestro plug-in en cuanto a otros paquetes u otros proyectos, las referencias que se utilizarán en tiempo de ejecución, las extensiones en las que se basa nuestro plug-in e incluso si servirá como punto de extensión para otro plug-in. También podremos configurar opciones de compilación.

Por otro lado, en el archivo plugin.xml encontramos una especificación más detallada de lo que será nuestro plug-in, como pueden ser las clases con las que irá relacionado, elementos de modelado a los que hará referencia, puntos de extensión a los que implementará, elementos sobre los que actuará, etc.

Por último, como todo proyecto de Eclipse, los proyectos de plug-in tienen su propia carpeta de archivos fuente, en la que podemos crear todo tipo de clases, interfaces, etc., que podrán ser utilizadas por el plug-in en cuestión.

2.2.2.2.- Extensiones

Las extensiones, son el método que nos proporciona Eclipse para añadir todo tipo de funcionalidades a nuestros proyectos, o incluso a nuestro IDE. Las extensiones se definen dentro de los plug-in, y se configuran como uno más de estos proyectos.

Más adelante encontraremos una explicación más detallada sobre la creación de extensiones, ya que la herramienta que hemos desarrollado hace uso de este mecanismo.

2.2.3.- EMF (Eclipse Modeling Framework)

Eclipse Modeling Framework (EMF) [16][8] es un framework de modelado y una herramienta de generación de código para construir otras aplicaciones basadas en un modelo de datos estructurados. EMF además se apoya en la tecnología Ecore [16][8] de Eclipse para la generación de modelos de forma totalmente gráfica. Partiendo de la descripción del modelo, EMF proporciona herramientas y soporte runtime para producir un conjunto de clases Java que implementan el propio modelo, un conjunto de clases que permiten su visualización y edición basándose en comandos del modelo, y un editor básico.

Los modelos pueden ser especificados de varias maneras: usando anotaciones Java, herramientas de modelado como Rational Rose o Ecore, que es la propia herramienta de Eclipse, o documentos XML, y después ser importados a EMF.

EMF proporciona la base para otras muchas aplicaciones basadas en construcción de modelos. En concreto, resulta de gran utilidad para el desarrollo de nuestra herramienta, ya que EMF nos permite:

- Describir el modelo de datos a alto nivel. Por ejemplo podemos usar un diagrama de Rational Rose, un diagrama UML, XSD, interfaces Java anotadas, o el propio editor Ecore de Eclipse para construir un fichero ecore con el modelo de datos de nuestro sistema.
- Con el fichero ecore, podemos generar un fichero genmodel que a su vez puede ser usado para generar el código Java que describe nuestro modelo. Crea además un editor en línea de comandos y otro gráfico como plugins de Eclipse. Todo eso sin tener que escribir una línea de código.
- Con este código generado, EMF nos permite serializar/deserializar nuestro modelo a XMI y XML, tener

relaciones entre objetos, notificaciones, notificaciones inversas, adaptadores, etc., todo sólo con la descripción a alto nivel de nuestro modelo y sin escribir prácticamente ni una sola línea de código.

- Podemos manipular el código generado, y si en un futuro hay cambios en el modelo, se puede recargar y regenerar el código, respetándose nuestros cambios.
- Podemos usar los ficheros `ecore` y `genmodel` anteriores para crear un editor gráfico apoyándose en otras tecnologías de Eclipse.

2.2.4.- GEF (*Graphical Editing Framework*)

GEF es un framework de Eclipse que permite al desarrollador crear editores visuales partiendo del modelo de una aplicación existente [2][5][8]. El framework se divide en dos plug-ins: Draw2d, que proporciona las herramientas necesarias para la creación de componentes gráficos y diseño de los mismos, y GEF, basado en la arquitectura MVC (Modelo-Vista-Controlador), y que trabaja por encima de Draw2d. A continuación, se explicará el funcionamiento de estos dos plug-ins y la manera en la interactúan entre ellos para la producción de editores visuales.

- *Draw2d*: como se dijo anteriormente, Draw2d es un conjunto de herramientas “lightweight” (peso ligero) para el manejo de componentes gráficos, que en adelante llamaremos *figuras*. Las figuras son simplemente objetos java, y no se corresponde con ningún recurso en el sistema operativo. Las figuras pueden ser construidas mediante una estructura padre-hijo, como se ve en la Figura 2.4.

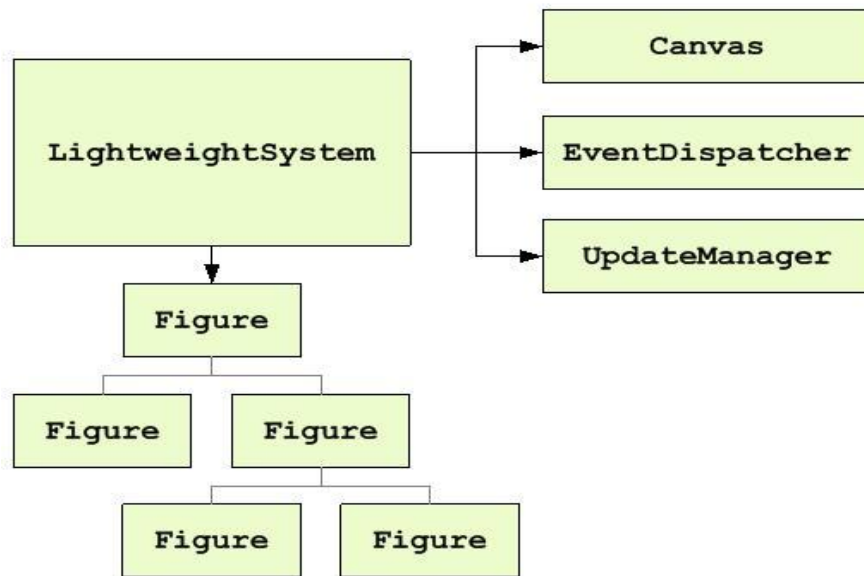


Figura 2.4 Estructura de Draw2d

El LightweightSystem asocia a cada composición de figuras un Canvas, lugar donde serán dibujadas. Además proporciona un medio de conexión entre las figuras y los eventos asociados a ellas mediante el EventDispatcher. Por último, el UpdateManager se encarga de coordinar las modificaciones realizadas en la presentación de las figuras, ya sean cambios de tamaño o de apariencia.

Draw2d nos proporciona, por tanto, todo tipo de clases y métodos para la creación y edición de figuras, es decir, de componentes gráficas para el editor.

- *GEF*: es el plug-in de edición propiamente dicho. GEF facilita la representación gráfica de cualquier modelo mediante el uso de figuras Draw2d. Además soporta interacciones con el usuario mediante ratón y teclado, así como con el Workbench de Eclipse.

La Figura 2.5 muestra una descripción, a bajo nivel, del funcionamiento de GEF.

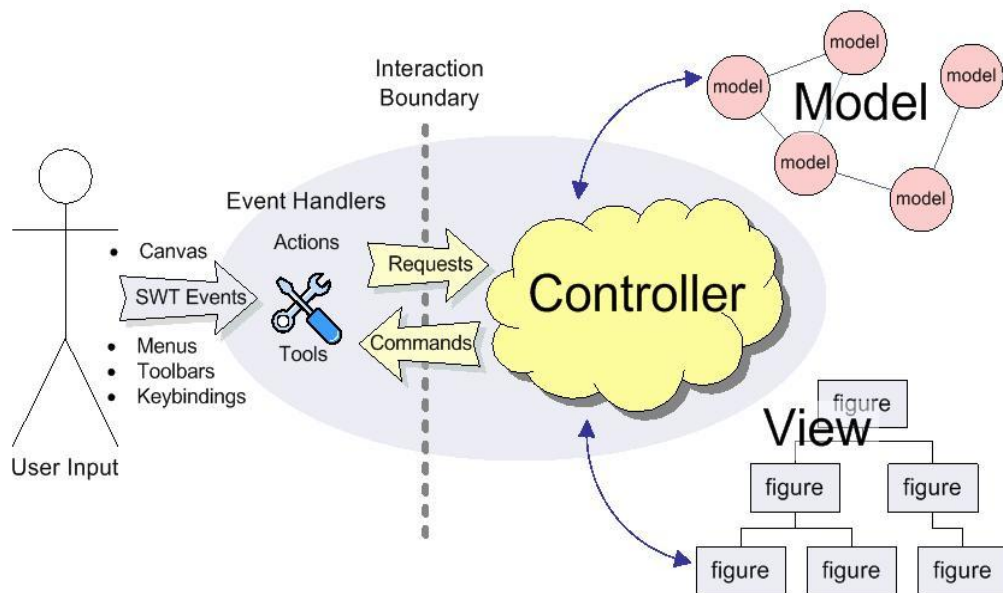


Figura 2.5 Descripción del funcionamiento de GEF

Podemos considerar GEF como la región situada en la parte central de la imagen. El framework proporciona una conexión entre el modelo y la vista (figuras), además de un conjunto de herramientas que permiten al usuario interactuar tanto con una parte como con otra. Pasamos a analizar cada una de las partes por separado:

- *Modelo*: es la base de GEF. La información que guarda es persistente, es decir, queda almacenada de forma permanente. El modelo debe proporcionar mecanismos para su modificación por parte del usuario.
- *Vista*: es cualquier elemento visible para el usuario. Como elementos visuales usaremos las figuras de Draw2d.
- *Controlador (EditParts)*: GEF establece un controlador para cada objeto del modelo. Estos controladores se conocen con el nombre de EditParts, y son la conexión entre el modelo y la

vista. Los EditParts se ayudan de otro componente llamado EditPolicy para la mayoría de las tareas de edición.

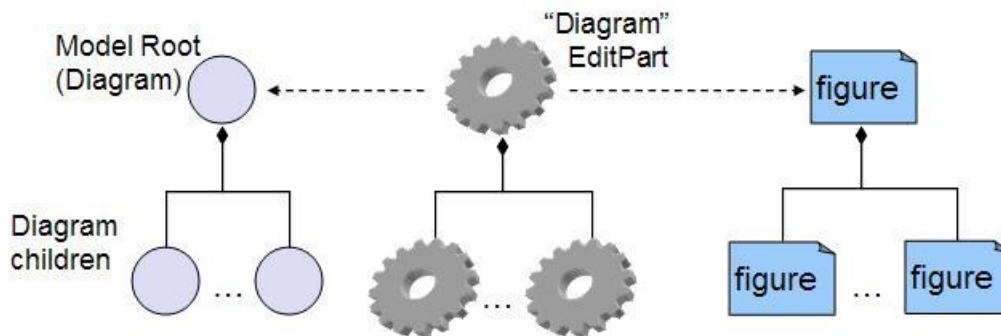


Figura 2.6 Estructura de las Editparts

En la Figura 2.6 podemos observar cómo las EditParts conforman una estructura similar a la del modelo, la cual se traslada también a la jerarquía de figuras dentro de la vista.

Sin embargo, existen excepciones a esta estructura arborescente. Estas excepciones son las conexiones. Una conexión representa una asociación entre dos nodos, la cual será representada en la vista como una conexión de Draw2d. El EditPart de una conexión es también un caso particular, puesto que está manejada por la EditPart del nodo origen y destino a los que conecta.

Así pues, podemos concluir diciendo que la principal responsabilidad de un EditPart es la creación y mantenimiento tanto de su propia vista, como la de sus hijos, como la de sus conexiones asociadas, así como del mantenimiento de su objeto en el modelo correspondiente.

2.2.5.- GMF (*Graphical Modeling Framework*)

Graphical Modeling Framework (GMF) [5] proporciona una herramienta para la generación de editores gráficos basados en EMF y GEF. Este último es el que permite al desarrollador crear un editor gráfico completo de forma rápida a partir de un modelo de una aplicación. Durante la utilización de GMF para la generación de un modelo, la descripción del modelo se realiza una sola vez, al comienzo. Una vez realizada la especificación del dominio, la herramienta se encarga de interpretar las correspondencias con el modelo durante el resto de proceso.

Una característica destacable en GMF es la reutilización de la definición gráfica para diferentes dominios y aplicaciones: esto es, se pueden reutilizar las metáforas gráficas ya definidas para conceptos de diferentes dominios y aplicaciones. Esta característica se consigue modelando por separado las componentes gráficas que se corresponden con cada uno de los elementos del dominio y la definición de la paleta de herramientas, la cual tendrá una herramienta por cada primitiva. Para completar el proceso de generación de un editor gráfico de dominio, GMF proporciona una definición de mapping o correspondencia mediante la que se asocia cada primitiva de modelado con su componente gráfica y con su herramienta dentro del editor que se está generando.

El desarrollo de un editor gráfico con GMF se basa en el diseño de seis modelos diferentes, los cuales pasamos a describir brevemente:

- *Modelo Ecore*: creado a partir de EMF.
- *Modelo Genmodel*: obtenido automáticamente a partir del modelo Ecore, contiene la información necesaria para generar el código referente al modelo y a la edición de éste.

-
- *Modelo Gmfgraph*: contiene toda la información gráfica relativa al editor. En este modelo especificaremos la representación tanto de las figuras como de las conexiones. Generalmente crearemos una figura o conexión por cada elemento del modelo ecore.
 - *Modelo Gmftool*: se encarga de la especificación de la paleta del editor. Por norma general, tendremos un elemento en la paleta por cada figura y conexión que hayamos creado en el modelo gmfggraph.
 - *Modelo Gmfmap*: podríamos considerarlo como el modelo más importante de los seis. Establece la relación entre todos los modelos anteriores, es decir, hace corresponder a cada objeto del modelo ecore, con su representación gráfica del modelo gmfggraph, y con su elemento de la paleta correspondiente al modelo gmftool. Además, en el gmfmmap estableceremos si un elemento del modelo ecore lo mapearemos como un nodo o una conexión.
 - *Modelo Gmfgen*: el modelo Gmfgen se crea automáticamente a partir del modelo gmfmmap y gmfmgenmodel. Relaciona estos dos modelos como último paso antes de la generación del código del editor.

Gracias a esta estructura basada en modelos, GMF proporciona un método eficiente a la par que sencillo para la creación de editores gráficos basados en la arquitectura de GEF y EMF.

3.- La herramienta de autor para <e-Tutor>:

3.1.-Notación gráfica:

El proyecto que hemos desarrollado es un editor gráfico para la creación de tutoriales. El entorno de trabajo que utilizaremos para ello está formado por una paleta en la parte derecha, en la que aparecen una sucesión de elementos para la creación, y por una ventana de edición, inicialmente en blanco, donde colocaremos la figura deseada en cada momento. La Figura 3.1 muestra un ejemplo del entorno de trabajo.

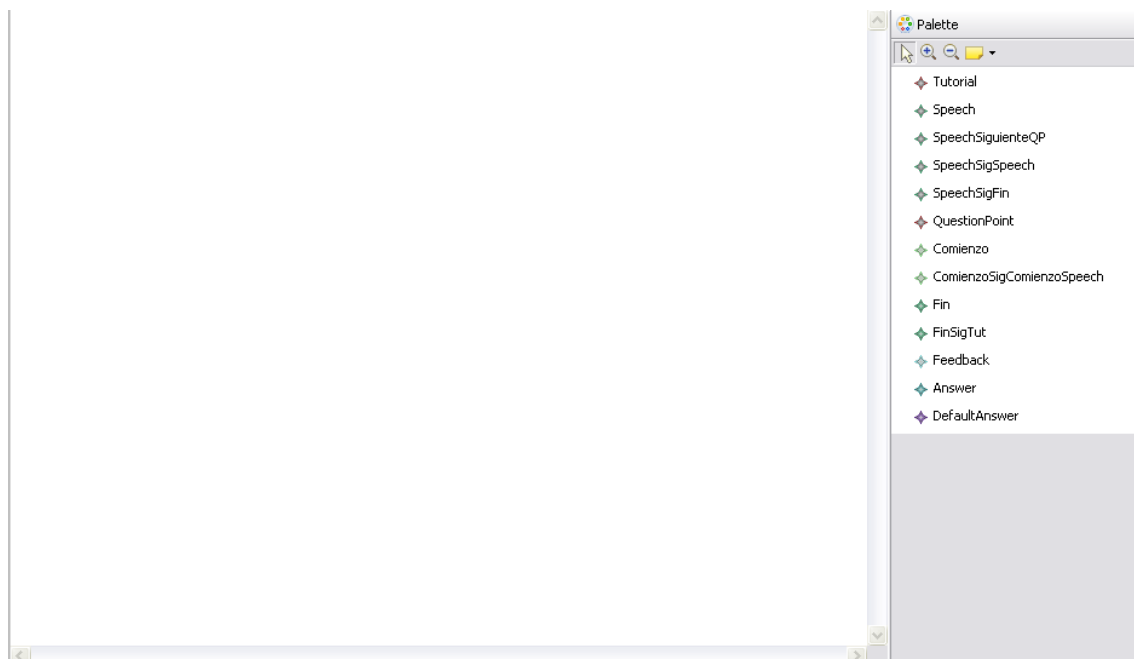


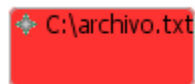
Figura 3.1 Ejemplo del entorno de trabajo

A continuación, listamos todos los elementos junto con su representación gráfica:

-
- *Comienzo*: indica el inicio del tutorial. La figura que lo representa es un círculo blanco con un borde negro.



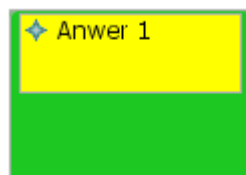
- *Speech*: representa cada bloque de texto que introducimos en el tutorial. Su figura es un cuadrado, con los bordes redondeados, rojo, y con un label o etiqueta, en el que aparece la ruta del archivo asociado a cada speech. En este archivo escribiremos el texto del speech.



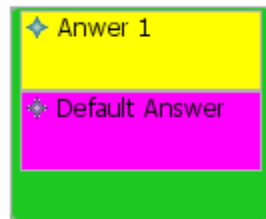
- *QuestionPoint*: elemento que representa una pregunta y contiene las diferentes respuestas. La figura correspondiente es un cuadrado verde.



- *Answer*: respuesta asociada a una pregunta. Se representa mediante un cuadrado amarillo, y, a diferencia de las demás, no se coloca en la ventana de edición, sino dentro de un QuestionPoint. También contiene un label con el texto de la respuesta.



- *DefaultAnswer*: respuesta por defecto asociada a una pregunta. La única diferencia gráfica con Answer es el color de la figura que lo representa, que en este caso es morado.



- *Fin*: elemento que delimita el final del editor. La figura que lo representa es un círculo negro.



- *Tutorial*: representa un nuevo tutorial anidado al que estamos editando. Su figura asociada es un círculo azul.



- *Feedback*: es una conexión especial del diagrama. Conecta una respuesta con un Speech. Lleva asociado un label que indica el orden en el que deben de aparecer los textos asociados a una respuesta. Se respresenta mediante una linea gris.



- El resto de elementos de la paleta son conexiones que permiten unir unas figuras con otras. La representación para todas ellas es una flecha gris.



La Figura 3.2 muestra un ejemplo de tutorial completo.

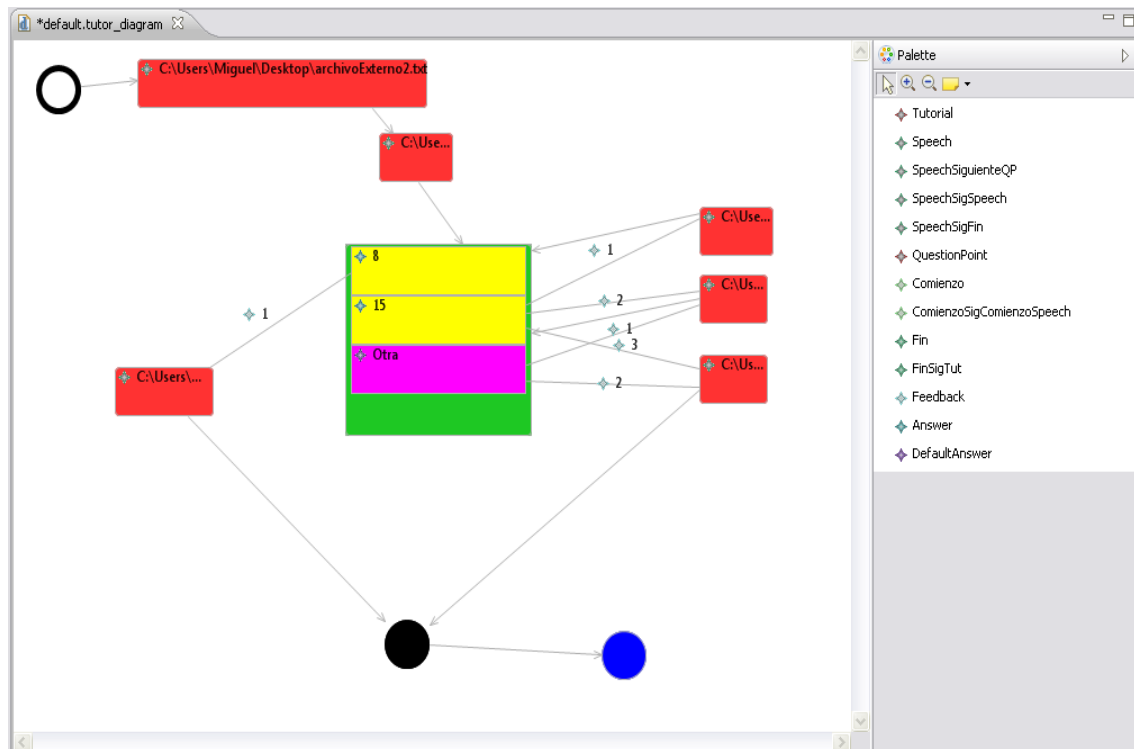
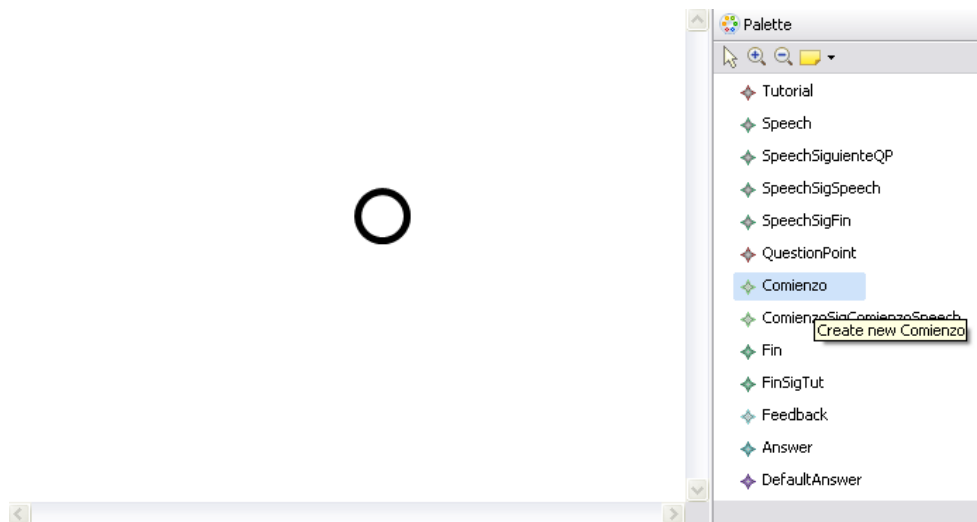


Figura 3.2 Ejemplo de tutorial completo

3.2.-La herramienta:

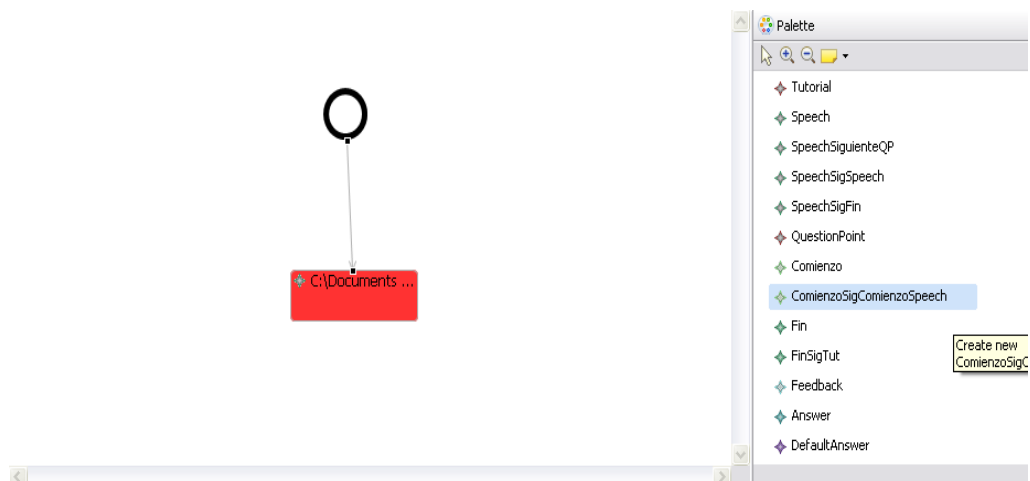
En este apartado se detallará el proceso de creación de un tutorial. El editor es una secuencia de figuras que representan la estructura que seguirá el tutorial creado por el usuario. Los elementos que constituyen el editor y el papel que desempeñan son:

- El elemento inicial del editor siempre debe ser una figura denominada Comienzo, que irá unida a otra llamada Speech mediante la conexión ComienzoSigComienzoSpeech. Para añadirla al editor, el usuario debe seleccionar el icono Comienzo de la paleta y acto seguido hacer click en la localización de la ventana de edición donde desee añadirlo.



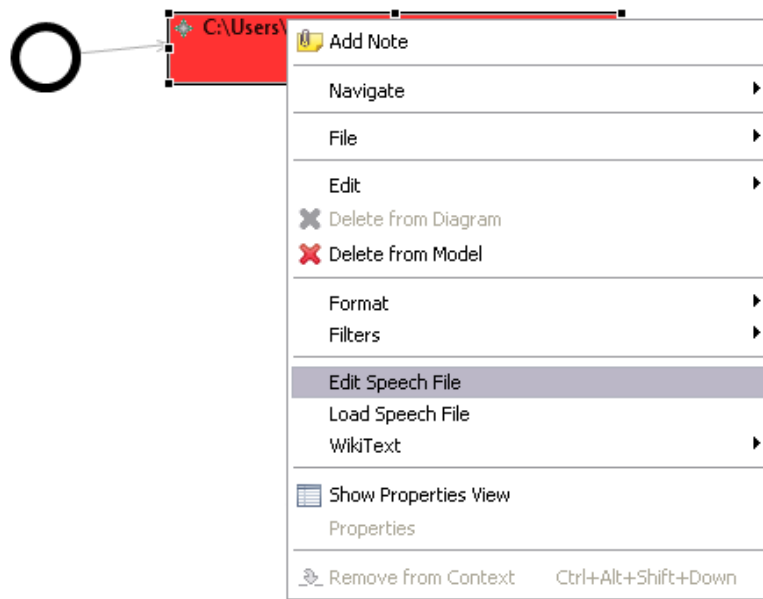
Para añadir la figura del elemento Speech, el usuario deberá realizar el mismo proceso, seleccionar el icono de la paleta y hacer click en la localización donde desea situarlo.

Para unir ambas figuras mediante ComienzoSigComiezoSpeech, debemos seleccionar esta conexión en la paleta y hacer click en Comienzo que se corresponde con el origen de ésta, y después en el Speech que deseamos, creándose de esta forma una flecha que unirá ambas figuras.



- El elemento Speech tiene un fichero asociado por defecto, cuya ruta se muestra en la figura del editor. En este archivo se

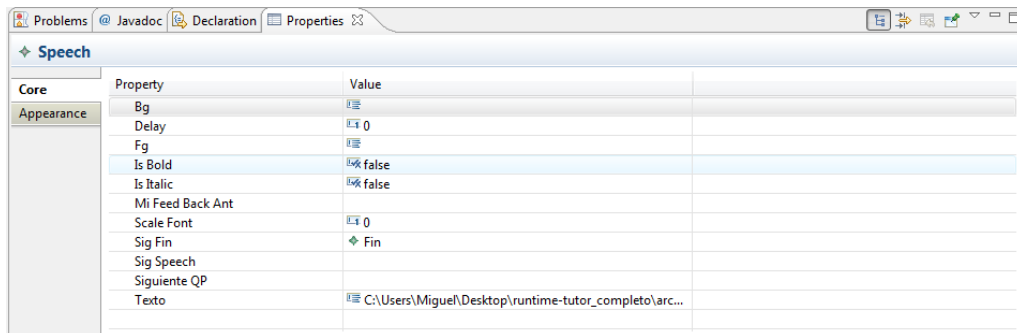
escribe y se guarda el texto que posteriormente aparecerá en el tutorial. Para abrirlo y editarlo, debemos hacer click derecho en la figura y seleccionar la opción Edit Speech File. Se nos abrirá en una nueva pestaña un editor de texto para poder modificarlo.



Existe también la opción de cargar un archivo ya existente, haciendo click derecho y seleccionando Load Speech File.

Cada Speech se puede conectar con tres figuras distintas: con otro Speech, con un QuestionPoint y por último con el elemento Fin, mediante las conexiones SpeechSigSpeech, SpeechSiguienteQP y SpeechSigFin, respectivamente.

Además los Speech tienen una serie de propiedades que podemos modificar. Para ello, abriremos la ventana de propiedades haciendo click derecho sobre la figura del Speech y seleccionaremos la opción Show Properties View.



Dentro de esta vista podemos modificar las siguientes opciones:

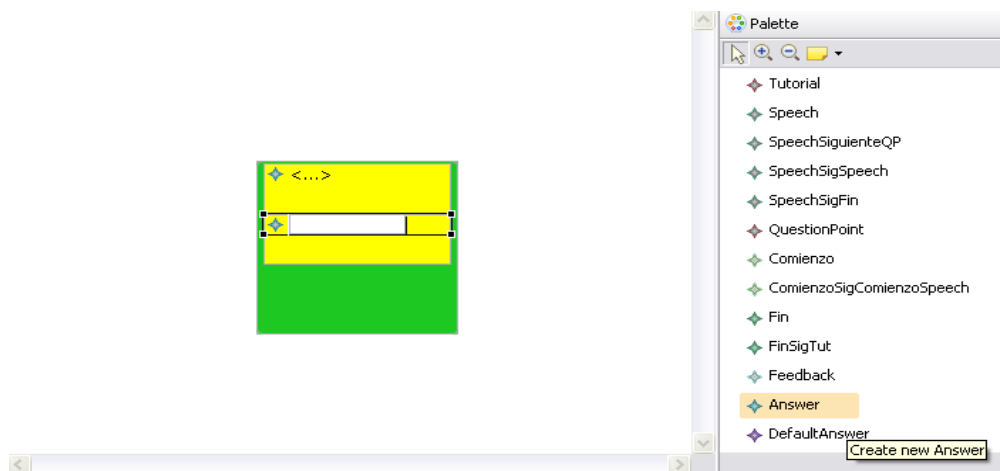
- *Bg*: cambia el color de fondo con el que aparecerá el texto del Speech durante la ejecución de tutorial.
- *Delay*: indica el tiempo que se mantendrá en pantalla el texto del Speech antes de pasar al siguiente elemento del tutorial
- *Fg*: representa el color que tendrá la letra del texto.
- *IsBold*: si activamos esta opción el texto aparecerá en negrita.
- *IsItalic*: convierte el texto a cursiva.
- *ScaleFont*: factor de escalado para la letra del texto. Cuanto mayor sea, más grande será el tamaño del texto.

Todos estos atributos llevan asociado un valor por defecto, es decir, no estamos obligados a modificarlos para que el tutorial funcione.

Por otro lado, no es recomendable modificar ninguno de los otros atributos que aparecen en esta ventana, ya que puede perjudicar al buen funcionamiento del tutorial.

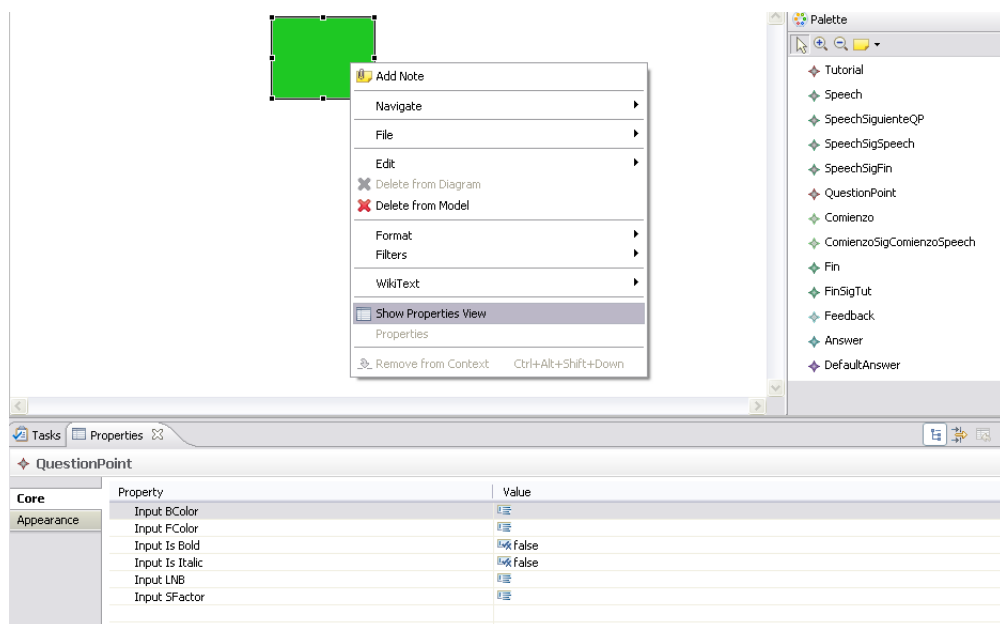
- Otro de los elementos que forman el editor es el denominado QuestionPoint, cuyo antecesor siempre será un Speech que contendrá el texto de la pregunta correspondiente. Dentro de cada QuestionPoint estarán las diferentes respuestas asociadas, incluyendo una respuesta por defecto.

Para añadir la figura QuestionPoint, debemos realizar el mismo proceso explicado anteriormente para Comienzo y Speech. A continuación, para introducir las respuestas debemos seleccionar Answer en la paleta y hacer click en el QuestionPoint. De esta forma, quedarán dentro de cada QuestionPoint sus posibles respuestas.

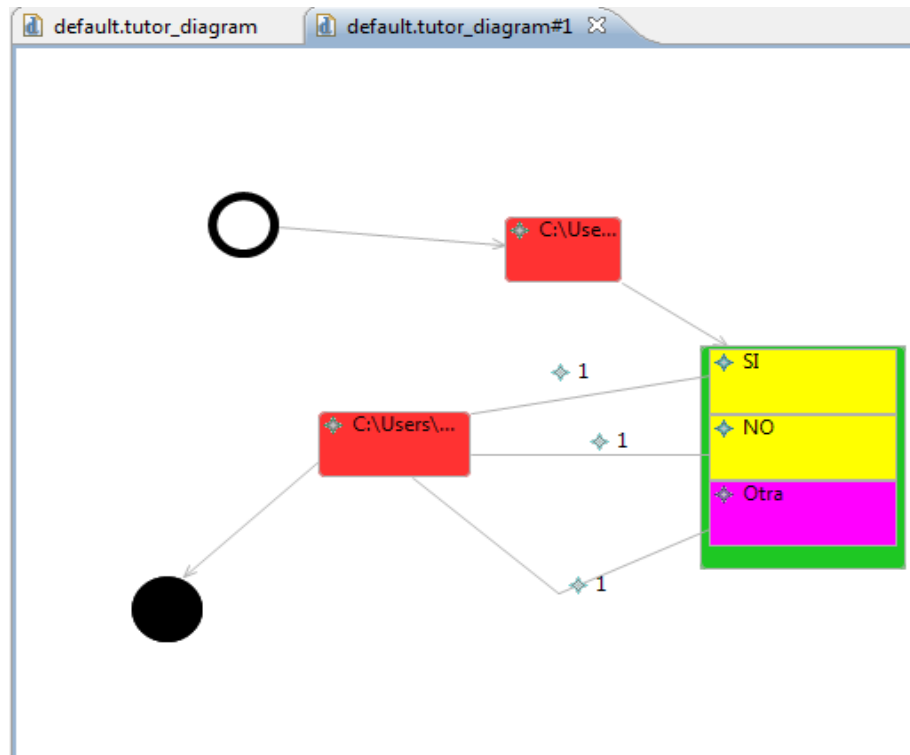


Si hacemos click derecho en la figura del QuestionPoint y seleccionamos la opción Show Properties View, se nos mostrará una lista de atributos que sirven para poder personalizar la parte del tutorial que se corresponde con la entrada por teclado. Los atributos que podemos modificar son los siguientes:

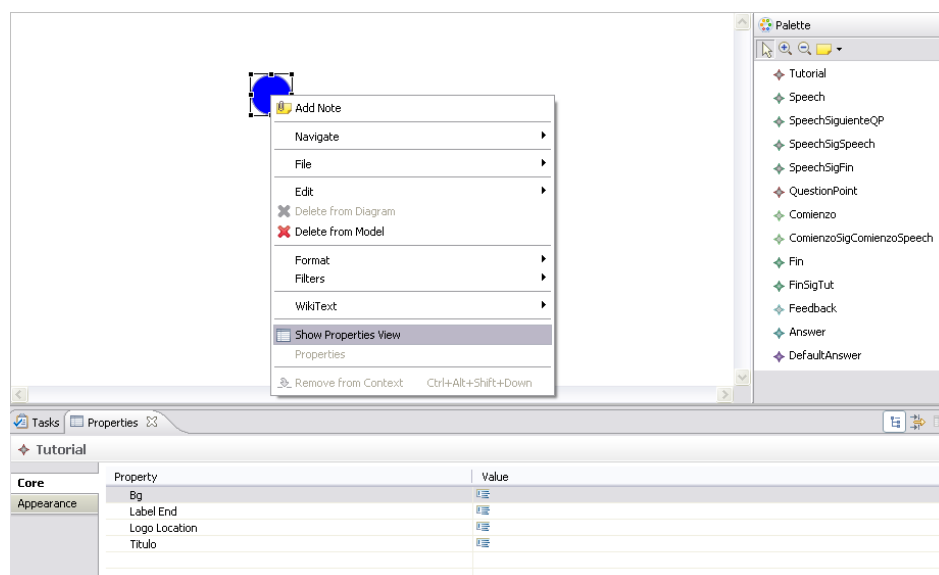
- *Input BColor*: permite cambiar el color de fondo de la entrada por teclado.
- *Input FColor*: cambia el color de la letra.
- *Input IsBold*: pone la letra en negrita.
- *Input IsItalic*: permite escribir en cursiva.
- *Input SFactor*: determina el tamaño de la letra.



- Las respuestas se representan mediante la figura Answer y la respuesta por defecto mediante DefaultAnswer. Ambas siempre deben ir situadas dentro de un QuestionPoint. Además se relacionan mediante la conexión Feedback con uno o varios Speech, que contendrán el texto que mostrará el tutorial en función de la respuesta introducida por el usuario. El Speech asociado a la respuesta por defecto, se mostrará en el caso de que la respuesta introducida no se corresponda con ninguna de las demás.
- Otro elemento importante del editor, es la conexión Feedback, que une un Answer con uno o varios Speech. Estos, se mostrarán dependiendo de la respuesta dada por el usuario y el número asociado a dicha conexión, que corresponde con el número de veces que se ha introducido la respuesta. Para añadirlo el usuario deberá seleccionarlo en la paleta, hacer click en el Answer al que corresponde y después en el Speech con el que lo queremos relacionar.



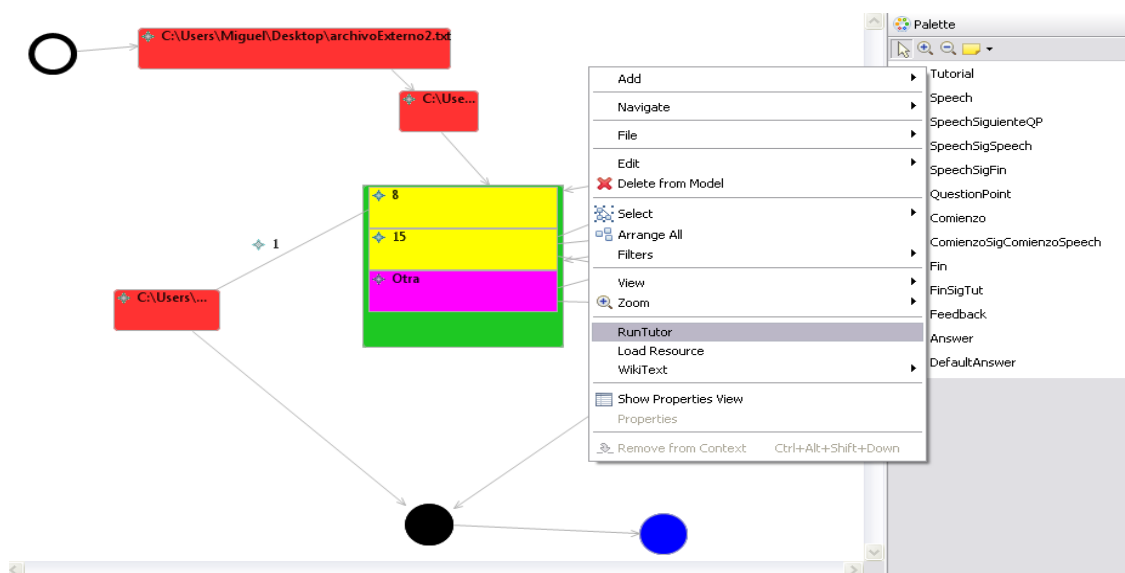
Este elemento también tiene opciones configurables, que podemos visualizar haciendo click derecho en el objeto y seleccionando Show Properties View.



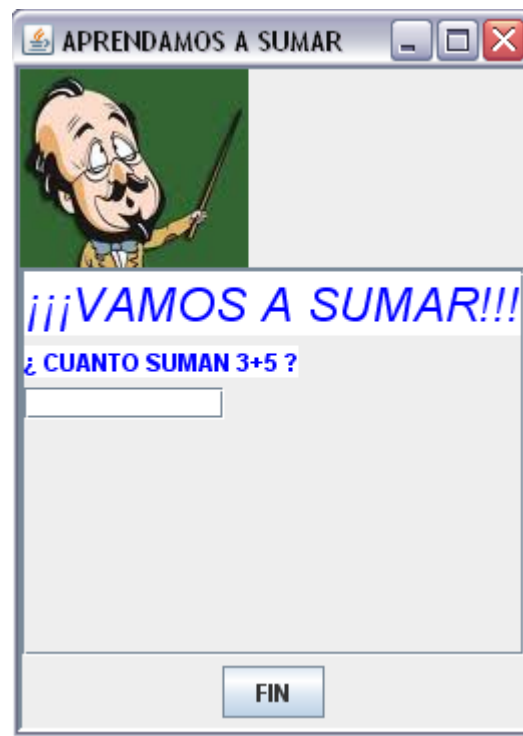
Estas propiedades nos permiten ponerle un Título al tutorial, cambiar el color de fondo con la opción Bg, asignarle un nombre al botón de salida del tutorial con Label End y con Logo Location podemos poner una imagen al inicio del tutorial, introduciendo como valor la ruta donde se encuentra. Estas propiedades también podemos modificarlas haciendo click en cualquier parte en blanco de la ventana de edición y mostrando la pestaña de propiedades.

En la Figura 3.2 se muestra un ejemplo de tutorial completo.

Para ejecutarlo, debemos hacer click derecho en cualquier punto de la ventana de edición, y a continuación seleccionar la opción RunTutor. Si lo hacemos desde la ventana principal se ejecutará el tutorial y todos los tutoriales que hayamos anidado posteriormente. Sin embargo, si seleccionamos RunTutor desde alguna de las otras pestañas de los tutoriales anidados, la aplicación tomará como primer tutorial el de la ventana en la que nos encontremos, olvidándose de los tutoriales padres.



Una vez hecho esto se nos abrirá una nueva ventana, con el tutorial que hemos creado.



4.- Desarrollo e implementación del proyecto.

En este apartado describiremos el desarrollo de la herramienta <e-Tutor>GE y la manera en la que se ha llevado a cabo su gestión. Se explicará la metodología de trabajo seguida, así como las distintas etapas en la vida de nuestro proyecto.

4.1.- Metodología de desarrollo:

Hemos seguido un método de trabajo similar al modelo de desarrollo "Scrum" [11]. Consta de tres fases:

- Una primera etapa breve de planificación, en la que se realiza una visión general del proyecto.
- Otra etapa de desarrollo, en la cual se realizan los denominados "Sprints". Un "Sprint" comienza con una reunión con el cliente, en este caso nuestro tutor del proyecto, y se decide qué funcionalidad se va a desarrollar hasta la próxima reunión. De esta manera, y mediante "Sprints" semanales, hemos logrado ir dando forma a nuestra herramienta <e-Tutor>GE.
- La última de las etapas es la de entrega y balance de los éxitos y fracasos.

La etapa de desarrollo se ha basado en reuniones semanales con el tutor, en las que se nos proponían objetivos a realizar para la semana siguiente. Asimismo, le mostrábamos los nuevos avances conseguidos respecto a la reunión anterior, y a la vez poníamos en común las dificultades con las que nos habíamos encontrado para llevar a cabo dichos avances.

En cuanto al método de trabajo, hemos procurado no hacer una división específica del trabajo, sino que todos los miembros hemos sido conscientes en todo momento del estado del proyecto, así como de todas las partes que lo conforman.

4.2.- Proceso de desarrollo del modelo:

El primer problema al que nos enfrentamos fue el de crear un metamodelo inicial que reflejara fielmente el comportamiento de nuestro sistema así como cada uno de sus elementos. Decimos *metamodelo inicial*, ya que durante la elaboración del proyecto lo hemos ido cambiando levemente, para adaptarlo a las exigencias que nos han surgido posteriormente.

Como ya es sabido, un modelo se describe usando conceptos que van más allá de clases y métodos. Para que pueda ser definido correctamente se necesita una terminología común para cada uno de los elementos. Dicha terminología la proporciona el *metamodelo*. Nuestro metamodelo será desarrollado en EMF. Además, para describirlo será también necesario un modelo: el *meta-metamodelo*. En Eclipse, este meta-metamodelo se llama Ecore.

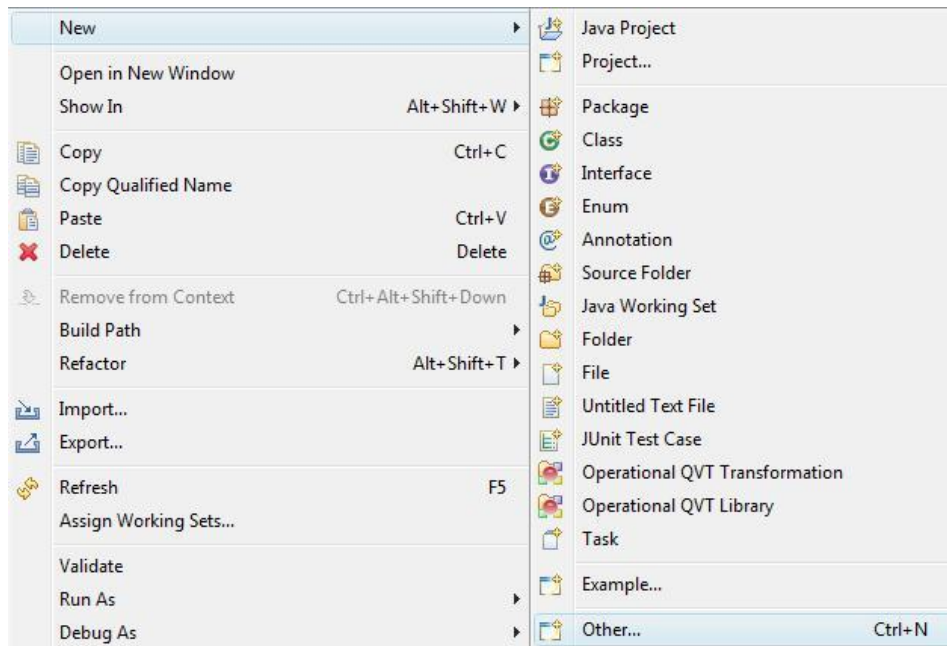
De esta forma, nosotros representaremos nuestro metamodelo de <e-Tutor> mediante un diagrama Ecore, es decir, un documento que nos permite especificar metamodelos Ecore. Además, al crear un diagrama Ecore, EMF genera al mismo tiempo un documento XML con extensión .ecore, que asocia al diagrama del metamodelo, y que es utilizado para la generación de código.

4.3.-Metamodelo <e-Tutor>GE:

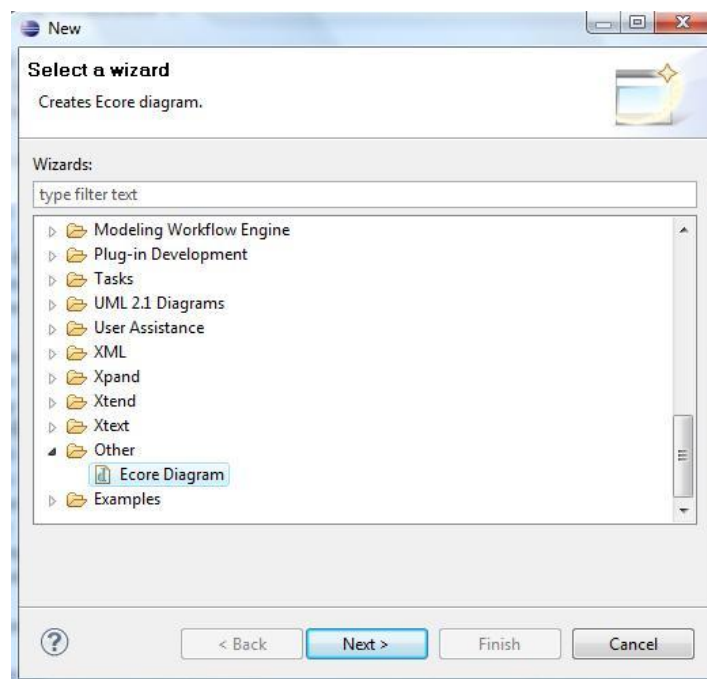
En Ingeniería del Software, y en general, en el desarrollo de software dirigido por modelos, un metamodelo es un modelo de nuestro lenguaje [7]. En nuestro caso partíamos con unos requisitos iniciales, ya que teníamos que realizar un editor gráfico para una herramienta ya creada (el ejecutor de tutoriales), por lo que las clases de nuestro metamodelo están pensadas para satisfacer las necesidades de <e-Tutor>.

El diagrama está creado a partir de la herramienta EMF, mediante la cual Eclipse nos da la posibilidad de diseñar nuestros propios metamodelos. Para ello, desde la ventana principal de Eclipse creamos un nuevo proyecto, seleccionando: *New->Project->Eclipse Modeling Framework->Empty EMF Project*.

Una vez creado el proyecto tenemos que crear el diagrama. Dentro de la carpeta "model" del proyecto creado hacemos click derecho, nos movemos a *New*, y aparecerá un menú contextual, en el que deberemos elegir la opción *Other*.



A continuación, se nos mostrarán una serie de opciones para elegir, y nuevamente habrá que ir sobre la opción *Other* y seleccionar *Ecore Diagram*. Tras elegir un nombre para el diagrama, Eclipse crea automáticamente un fichero *.ecore*, que será el reflejo de nuestro diagrama.



Una vez creado el diagrama, lo único que queda es dibujar nuestro modelo, en función de las características del sistema que queramos representar.

La paleta de herramientas que nos proporciona EMF incluye los diferentes elementos y relaciones que nos permiten definir un modelo Ecore. En la paleta se puede distinguir las siguientes herramientas:

- *Eclass*: con la que podremos crear nuevas clases.
- *Epackage*: con la que crearemos nuevos paquetes.
- *EAnnotation*: con la que se crearán anotaciones y comentarios.
- *EDataType*: con la que crear nuevos tipos de datos.
- *EAttribute*: con la que agregar atributos a las clases.
- *EOperation*: con la que añadir operaciones a las clases.
- *Association*: Relación de asociación.
- *Aggregation*: Relación de agregación.
- *Generalization*: Herencia.

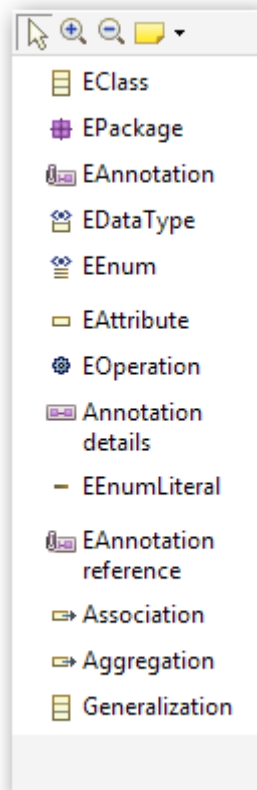


Figura 4.1 Paleta de Ecore

La construcción del modelo, usando cada una de estas herramientas, se realiza mediante “drag and drop”. Esto significa que, para añadir una nueva clase, primero pulsamos sobre la herramienta EClass de la paleta y a continuación, en cualquier parte del tapiz de modelado. El nombre de una clase en Eclipse debe comenzar siempre con una letra mayúscula. Además, se nos permite añadir a esta clase tantos atributos y operaciones como se deseen. Hay dos posibilidades de hacerlo:

- La primera es mediante el menú emergente que aparece superponiendo el puntero sobre la clase en cuestión.
- La otra es mediante las herramientas de la paleta, arrastrando y soltando dentro de la clase sobre la que se deseen añadir estos atributos u operaciones.

Para instanciar las relaciones, disponemos de las tres últimas herramientas de la paleta. Para relacionar dos clases de nuestro modelo, debemos seleccionar la relación que queramos en la paleta y a continuación seleccionar la clase desde la que parte nuestra relación (en el caso de herencia, la clase hija, y en el caso de la agregación, la clase compuesta) y después arrastrar y seleccionar la clase destino de la relación (en el caso de herencia, la clase padre, y en el caso de la agregación, la clase componente). Hay que tener en cuenta que las asociaciones en EMF son siempre unidireccionales. Si queremos modelar una asociación bidireccional lo que tenemos que hacer es crear dos asociaciones que relacionen ambas clases en sentidos opuestos y fusionarlas en una sola bidireccional. Para ello, se ha de seleccionar dentro de una de las asociaciones su opuesta en la propiedad Eopposite.

Para definir el tipo de un atributo, clase o relación es necesario editar sus propiedades. Esto se realiza pulsando con el botón derecho encima del elemento del que se desean ver las propiedades y seleccionar la opción Show Properties View. A continuación, aparece una ventana en la parte inferior del entorno de Eclipse mediante la que se pueden editar todas las propiedades de cada elemento. Para editar el tipo de los atributos, se ha de pinchar en la propiedad EType y seleccionar el tipo de datos adecuado para cada uno de ellos.

Dado que EMF expresa en formato XML el modelo correspondiente a un diagrama ecore, conviene resaltar que para definir un modelo correctamente en EMF es necesario incluir un elemento raíz (una clase) que relacione mediante agregación todas las clases del modelo, en nuestro caso de estudio la clase Tutorial.

En la Figura 4.2 se muestra nuestro metamodelo completo. En la Figura 4.3 aparece el correspondiente modelo.ecore, a partir del cual,

y ayudándonos del framework GMF explicado anteriormente, construiremos nuestro editor.

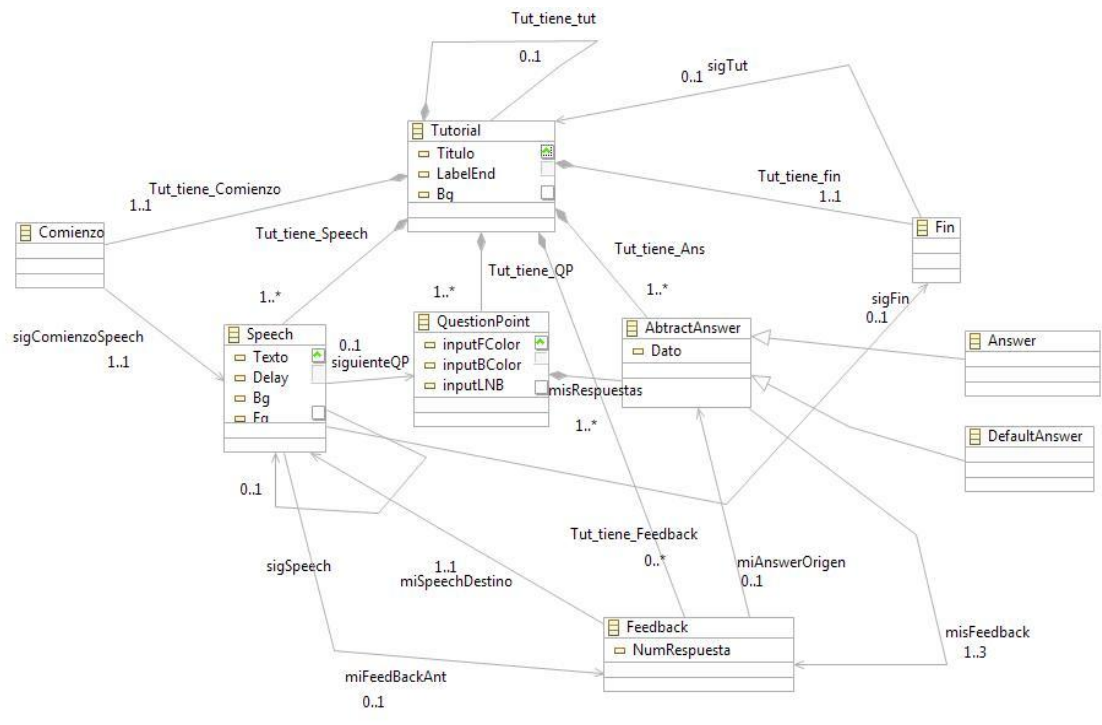


Figura 4.2 Metamodelo <e-Tutor> GE

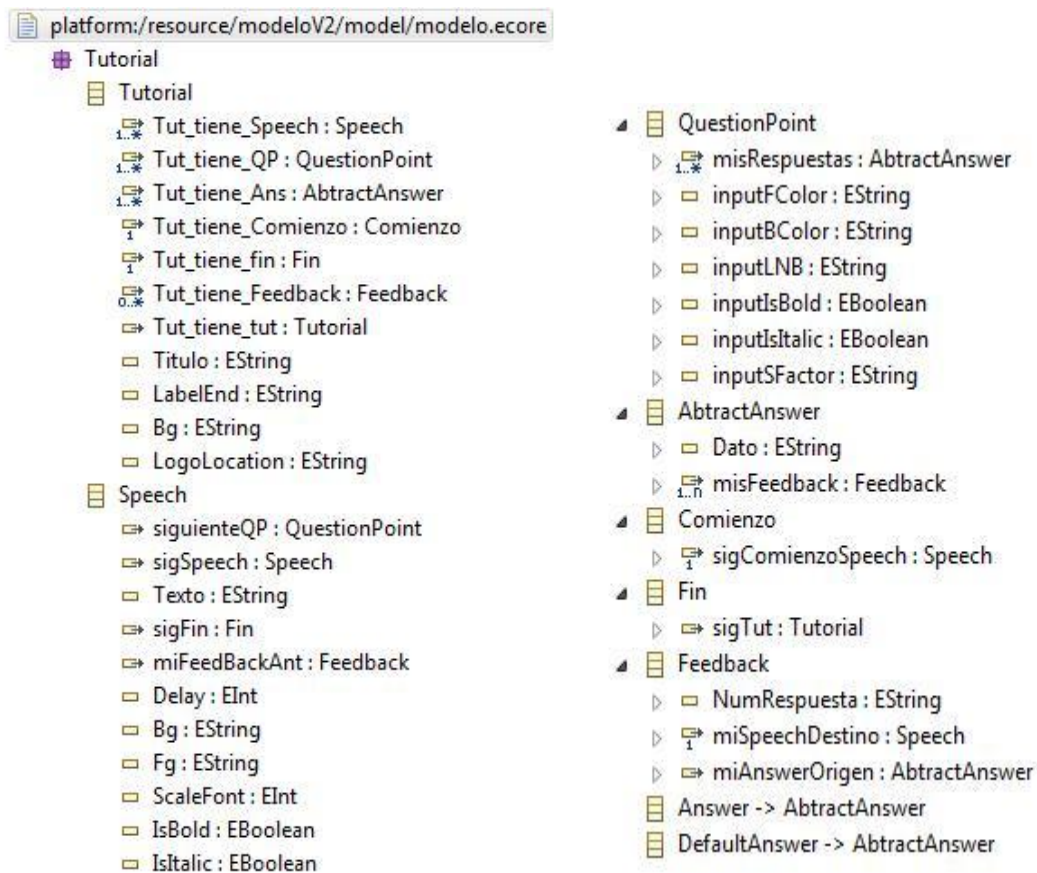


Figura 4.3 Metamodelo ecore <e-Tutor> GE

En nuestro metamodelo hay una clase principal denominada Tutorial, que contiene a todas las demás. Esta clase tiene varios atributos: Título (Título del tutorial), LabelEnd (Texto en el botón de salida del tutorial), Bg (Color de fondo) y LogoLocation (Imagen al inicio del tutorial). Las demás clases están relacionadas con ésta mediante una conexión de agregación. También existe una relación de agregación que tiene como origen y destino la propia clase Tutorial, lo que nos permite la creación de tutoriales anidados.

Cada una de las clases que contiene nuestro modelo son:

- *Comienzo*: tiene una relación de agregación con Tutorial, y una cardinalidad que obliga a que todo tutorial tenga un comienzo y sólo uno. Además, existe una relación de asociación con Speech, para ligar este elemento con su texto correspondiente.

Después de un comienzo siempre debe aparecer un Speech, ya sea un texto inicial o el enunciado de una pregunta.

- *Speech*: en un tutorial debe existir al menos un texto, por lo que su cardinalidad respecto a la relación de agregación es de uno a muchos, lo que nos permite introducir tantos Speech como se deseen. Existe también una relación de asociación con origen y destino en esta misma clase, por lo tanto podemos introducir un texto a continuación de otro.

Una relación con origen en esta clase y destino en la clase QuestionPoint, establece este texto como enunciado de una pregunta. Todo texto puede estar asociado a lo sumo a una pregunta.

- *QuestionPoint*: contiene los atributos que permiten personalizar la entrada por teclado en la que el usuario contesta a las preguntas. Estos atributos son: InputFColor (color de la fuente), InputBColor (color de fondo), InputLNB (label next button), InputIsBold (fuente en negrita), InputIsItalic (fuente en cursiva), InputSFactor (tamaño de fuente).

En un tutorial se pueden incluir varias preguntas, pero al menos debe existir una.

La clase AbstractAnswer tiene una relación de agregación a QuestionPoint, para asociar las posibles respuestas correspondientes a una pregunta concreta. Una pregunta puede tener varias respuestas asociadas.

- *AbstractAnswer*: tiene un atributo de tipo String, llamado Dato, que contiene el texto de la respuesta.

Una respuesta debe aparecer al menos una vez en un tutorial, asociada a una pregunta.

De esta clase sale una relación de asociación hacia la clase Feedback, para relacionar la respuesta con el texto que se

mostrará inmediatamente después de ella, dependiendo del número de veces que se haya repetido.

De ella heredan Answer y DefaultAnswer, que son los dos tipos de respuestas que existen.

- *Feedback*: es diferente a todas las demás porque representa una conexión, que tiene como origen una respuesta y como destino el texto que mostrará el tutorial a continuación. Consta de un atributo, NumRespuesta de tipo entero, que indica el número de veces que esa respuesta ha sido elegida por el usuario.

Puede aparecer en un tutorial un número ilimitado de veces.

- *Fin*: delimita el final de nuestro tutorial. Debe existir un final y sólo uno. Además, tiene una asociación con Tutorial, para poder anidar el tutorial que finaliza en esta clase con uno nuevo.

4.4.- Editor Gráfico <e-Tutor>GE:

El editor gráfico de la herramienta se ha desarrollado utilizando las posibilidades ofrecidas por Graphical Modeling Framework (GMF), que permite al desarrollador crear un editor gráfico completo de forma rápida a partir de un modelo de una aplicación. Todos y cada uno de los subprocesos de GMF están relacionados con el modelo y, como indica su nombre, Ecore constituirá el centro del proyecto en todo momento. El núcleo de GMF es el modelo de definición gráfica (Graphical Definition Model), que almacena información de los elementos gráficos que aparecen en el editor generado en tiempo de ejecución. Para la definición de la paleta de herramientas y otros elementos de la interfaz del editor se emplea un segundo modelo (Tooling Definition Model). La relación entre ambas partes se establece a través de un tercer modelo, conocido como modelo de

definición de mapeos (Mapping Definition Model). Finalmente, GMF dispone de un modelo generador (Generator Model) que permite definir los detalles de la implementación en la fase de generación del código Java del plug-in del editor.

A continuación, se procede a aplicar dicho proceso para la creación del editor.

4.4.1.- Creación de un proyecto GMF

Para crear un nuevo proyecto vamos a *File -> New -> Project* y en la carpeta *Graphical Modeling Framework* seleccionamos *New GMF Project*.

Le proporcionamos un nombre al proyecto y pulsamos *Next*. Es importante que el proyecto no se llame como ninguna de las clases ya que podría dar errores de compilación al generar el código.

Activamos la casilla *Show dashboard view for the created Project* para obtener una vista del estado de nuestro proyecto como la de la Figura 4.4. En caso de que se cierre esta vista, estará accesible desde *Window -> Show view -> Other -> General -> GMF dashboard*.

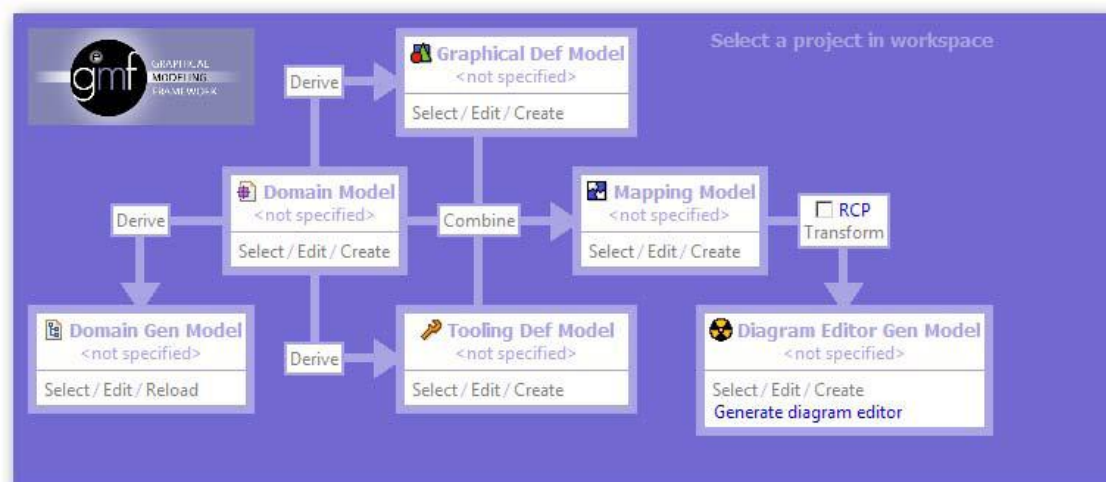


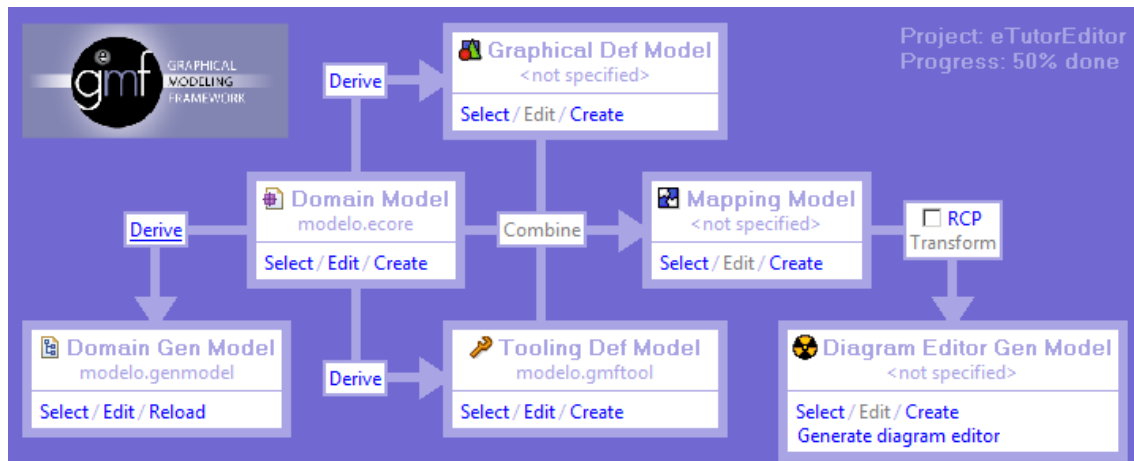
Figura 4.4 Dashboard de GMF

4.4.2.- Creación o importación de un modelo

El objetivo principal de GMF es proporcionar un editor gráfico para modelar diferentes entornos partiendo de un modelo o metamodelo de entrada. Dicho modelo es posible crearlo dentro de GMF desde cero o bien importar un modelo realizado previamente:

- Creación del modelo dentro del proyecto GMF: para crear el modelo pulsamos con el botón derecho sobre el proyecto, seleccionamos *New -> Other* y encontraremos el documento de tipo Ecore Diagram, pudiendo crear el modelo del mismo modo que se ha presentado en la sección anterior.
- Importación del modelo al proyecto GMF: no es necesario crear un modelo dentro del proyecto raíz GMF, sino que es posible importar uno previamente creado. Existen dos posibles maneras de hacerlo:
 1. Clicamos con el botón derecho sobre el proyecto GMF, utilizamos *New -> Other*, y en la carpeta Eclipse Modeling Framework seleccionamos EMF Generator Model. A continuación, señalamos la carpeta donde queremos introducirlo y le asignamos un nombre. Después, seleccionamos el tipo del modelo que vamos a importar; en nuestro caso será Ecore model. Una vez hecho esto, pulsamos *Browse Workspace*, si el modelo Ecore se encuentra en nuestro espacio de trabajo en un proyecto abierto, o *Browse File System* si el modelo está fuera del espacio de trabajo y pulsamos *Finish*. De esta forma, se nos habrá creado el modelo Genmodel asociado a Ecore y ambos aparecerán en la carpeta model del proyecto GMF.
 2. Copiamos y pegamos el modelo Ecore creado anteriormente dentro de la carpeta model de nuestro

proyecto GMF. Después, clicamos sobre la opción *Select* del cuadro Domain Model perteneciente a GMF Dashboard y seleccionamos el modelo dentro del proyecto GMF. Para generar el modelo Genmodel asociado a este, debemos hacer click en el cuadro de derivación que aparece a su izquierda, seleccionamos la carpeta model del proyecto, le asignamos un nombre y pulsamos *Next*. Al igual que en el modo anterior seleccionamos como tipo de archivo importado Ecore model y seleccionamos el modelo Ecore que se encuentra en nuestro proyecto.



4.4.3.- Generación del código del modelo

En el punto anterior se ha creado un nuevo elemento con extensión genmodel. Este elemento es un modelo que permite transformar automáticamente el modelo Ecore que hemos definido a código fuente. El resultado de la generación es un conjunto de clases Java, que serán utilizadas más adelante para que todos los elementos que constituyen la herramienta, se comporten tal y como el modelo ecore establece.

Antes de generar el código del modelo es preciso configurar el modelo Genmodel para especificar su paquete base. Para ello hacemos doble clic en modelo Genmodel. Se abre una ventana con un árbol similar al del modelo Ecore. Hacemos doble click sobre el paquete morado y se nos abrirá la ventana de propiedades, donde debemos asignar el mismo nombre a las propiedades Base Package, Prefix y Package.

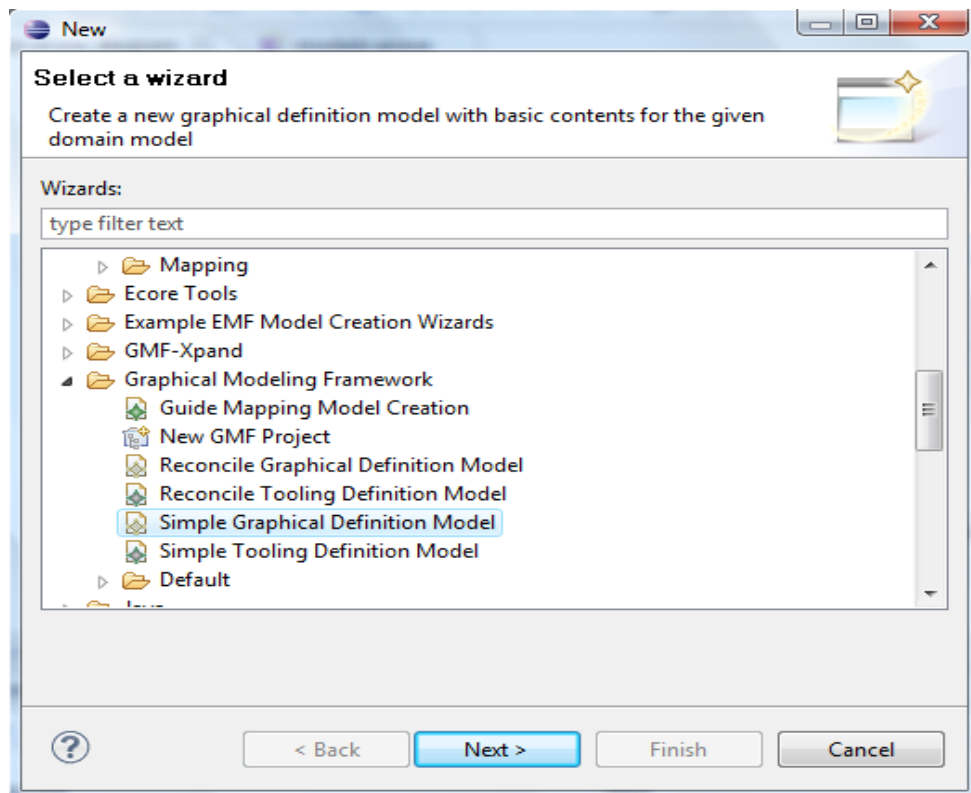
Ahora está todo listo para generar el código del modelo. Clicamos con el botón derecho sobre el paquete morado y seleccionamos *Generate Model Code*. Se generará automáticamente un plug-in con el código del modelo. A continuación, realizamos el mismo proceso seleccionando *Generate Edit Code* y *Generate Editor Code*. Este último será el intérprete de Eclipse que nos permitirá ver nuestro modelo específico de dominio en forma de árbol.

4.4.4.- Definición gráfica del modelo

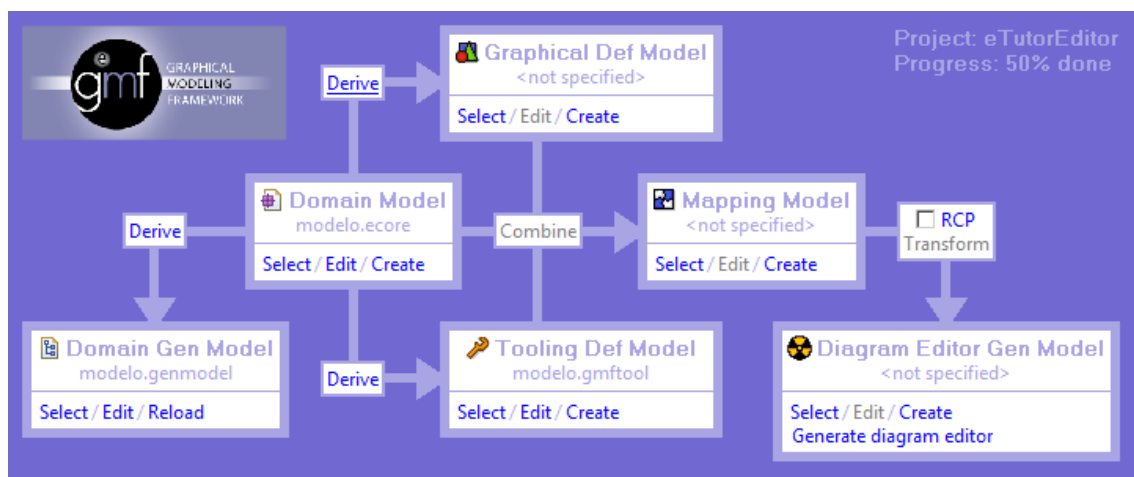
La definición gráfica del modelo consiste en definir el aspecto que van a tener las primitivas de modelado en nuestro editor gráfico. Así, GMF, de una forma totalmente automática, nos permite personalizar el aspecto de la aplicación de construcción de modelos específicos de dominio a nuestro gusto.

Debemos generar un archivo con extensión gmfgraph, para lo cual podemos seguir dos métodos distintos:

1. Hacemos click con el botón derecho sobre el proyecto GMF, utilizamos *New -> Other*, vamos a la carpeta *Graphical Modeling Framework* y seleccionamos *Simple Graphical Definition Model* y pulsamos *Next*.



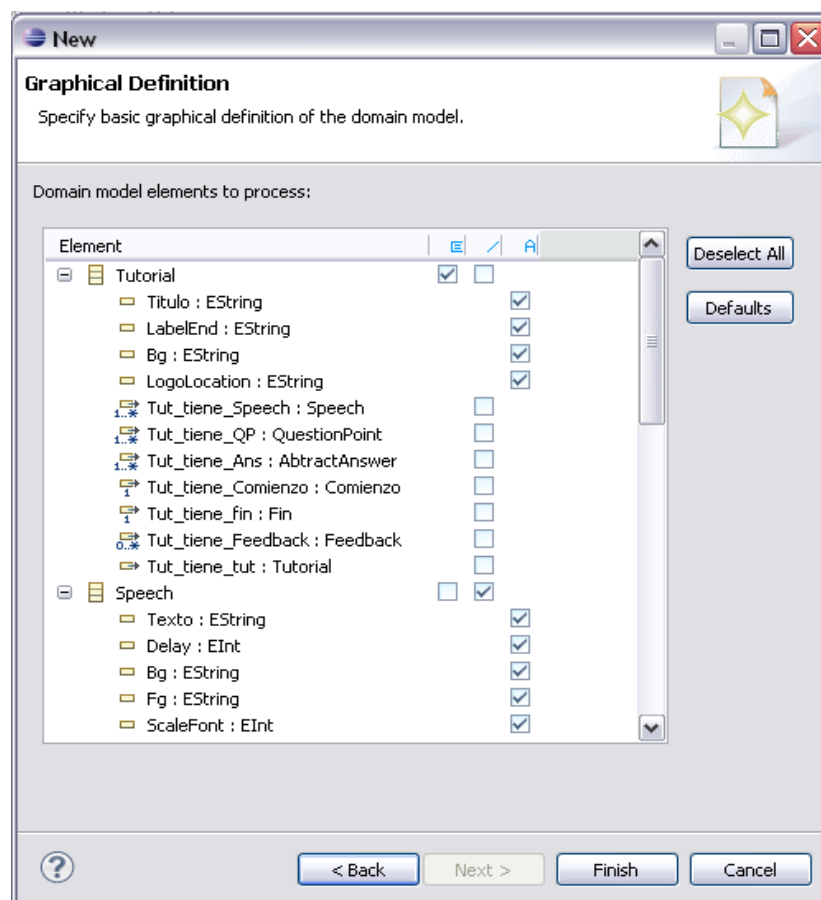
2. Utilizamos el DashBoard, que nos permitirá derivar este modelo a partir de nuestro modelo Ecore.



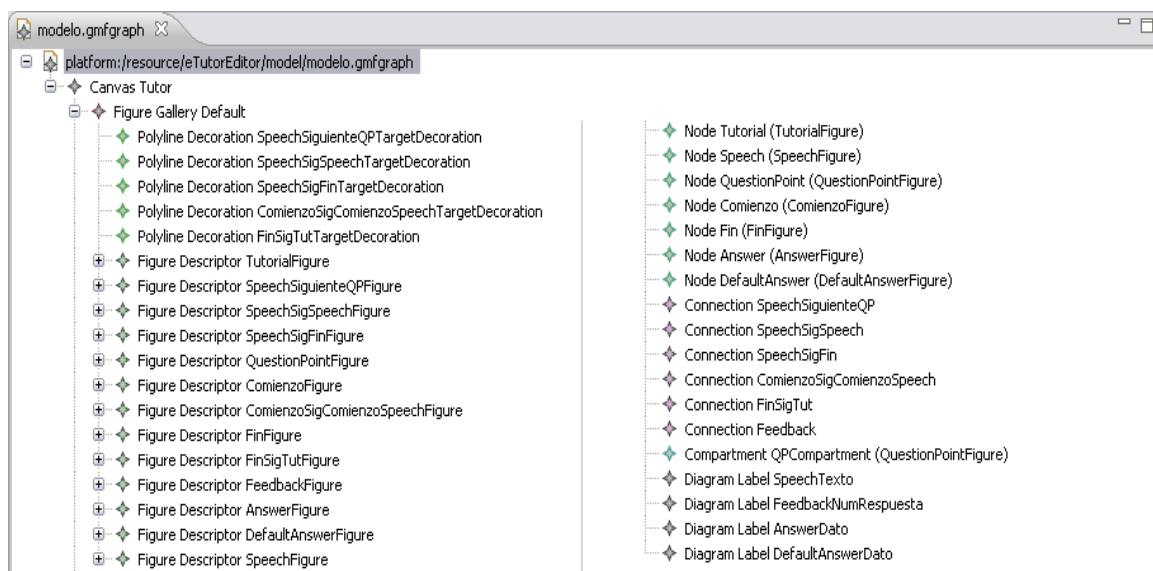
A continuación, seleccionamos la carpeta model de nuestro proyecto y le damos un nombre. En la siguiente ventana debemos relacionarlo

con nuestro modelo Ecore. En la ventana interior denominada Diagram Element, seleccionamos la clase principal que contiene a todas las demás (en nuestro caso será la clase Tutorial), y pulsamos *Next*.

Ahora podremos elegir qué elementos del modelo del dominio actuarán como nodos, cuáles como enlaces y cuáles como etiquetas. Se nos abrirá una ventana con una tabla de tres columnas en la que se encuentran colocados de izquierda a derecha, y están representados gráficamente en la cabecera de la siguiente forma: cuadrado con rayas horizontales (nodos), línea inclinada (enlaces), y letra A (etiquetas). Cada elemento del modelo tiene asociada una *checkbox* (casilla de selección) en cada columna de la tabla que es posible seleccionar, para así elegir el tipo de representación gráfica que se desea. Pulsamos *Finish*.



Al pulsar *Finish* aparecerá en la ventana el *modelo.gmfgraph*, que tiene estructura de árbol. Si desplegamos el elemento raíz, nos aparecerá una vista similar a:

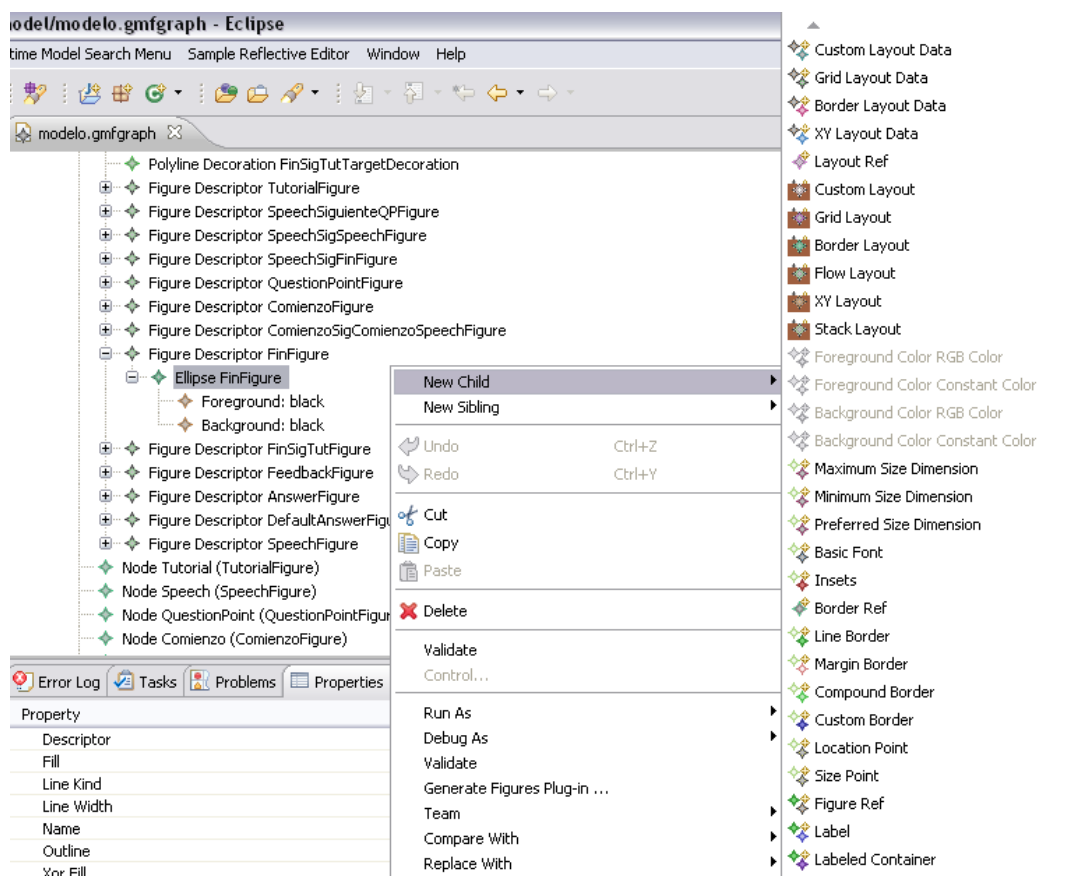


A continuación, seleccionamos *Figure Gallery Default* observando que los enlaces van determinados por un *Connection* y el nombre de la primitiva que representa ese enlace. Las entradas *Polyline Decoration* representan el adorno final que llevarán las conexiones, en nuestro caso, una flecha.

Los nodos del modelo vienen determinados por un descriptor de figura (*Figure Descriptor*) que contiene la figura *Rectangle* y las funciones de acceso a esas etiquetas *Child Access* *getFigure*. Si a su vez desplegamos la figura *Rectangle* de cada nodo, vemos que contiene sus etiquetas *Label* y la alineación de las mismas *Flow Layout*. Todas las propiedades de cada figura son editables. Si queremos cambiar la figura que representa a un nodo, debemos borrar la figura *Rectangle*, que nos aparece por defecto, hacer click derecho sobre el descriptor de figura, seleccionar *New Child* y se nos desplegará una lista de opciones donde podremos elegir la opción que

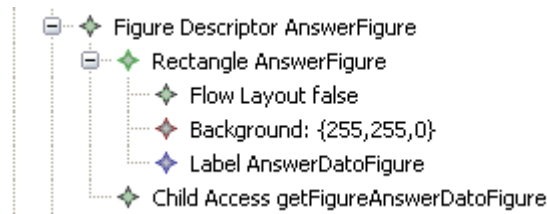
más se ajuste a nuestro editor. Para editar la figura de un enlace seguimos el mismo proceso, borramos la figura por defecto y añadimos la que deseemos.

Además, cada figura tiene propiedades configurables, si hacemos click derecho sobre ella y seleccionamos *New Child* se despliega una lista que contiene opciones como color de fondo, ancho del borde, etc. Podemos cambiar su valor en la ventana de propiedades.

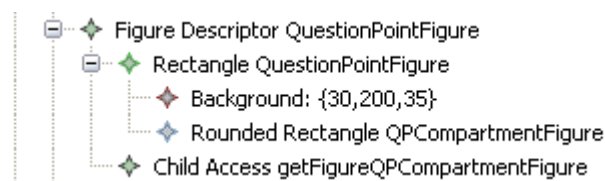


Los atributos que hemos configurado anteriormente como etiquetas aparecen en este esquema representados por Label. Para cada Label existe un *Child Access getFigure* que permite al usuario acceder a este y escribir texto en él. Para añadirlo clicamos con el botón derecho sobre el descriptor de figura y seleccionamos *New Child* ->

Child Access o bien hacemos click con el botón derecho sobre la figura y seleccionamos *New Sibling -> Child Access*.



Un caso especial es *QuestionPoint*, que a diferencia de las demás figuras contendrá las respuestas dentro de ella. Para ello, le añadimos una nueva figura como hijo a la que accedemos mediante un *Child Access*. De esta forma, el usuario puede introducir dentro de esta figura las respuestas y borrarlas.



A parte de esto, por cada figura existe un nodo *Node* de acceso al *FigureDescriptor*, por cada enlace una *Connection* y por cada Label un *Diagram Label*.

Cada *Node* debemos asociarlo en la ventana de propiedades con el descriptor de la figura correspondiente. Con *Connection* y *Diagram Label* hacemos lo mismo.

Además, para la figura creada dentro del *QuestionPoint* debemos añadir un nuevo elemento denominado *Compartment*, al que igualmente debemos asociar con el descriptor de figura y con el *Child Access* correspondiente.

Node Speech (SpeechFigure)

Property	Value
Affixed Parent Side	NONE
Content Pane	
Figure	Figure Descriptor SpeechFigure
Name	Speech
Resize Constraint	NSEW

Connection SpeechSiguienteQP

Property	Value
Figure	Figure Descriptor SpeechSiguienteQPFigure
Name	SpeechSiguienteQP

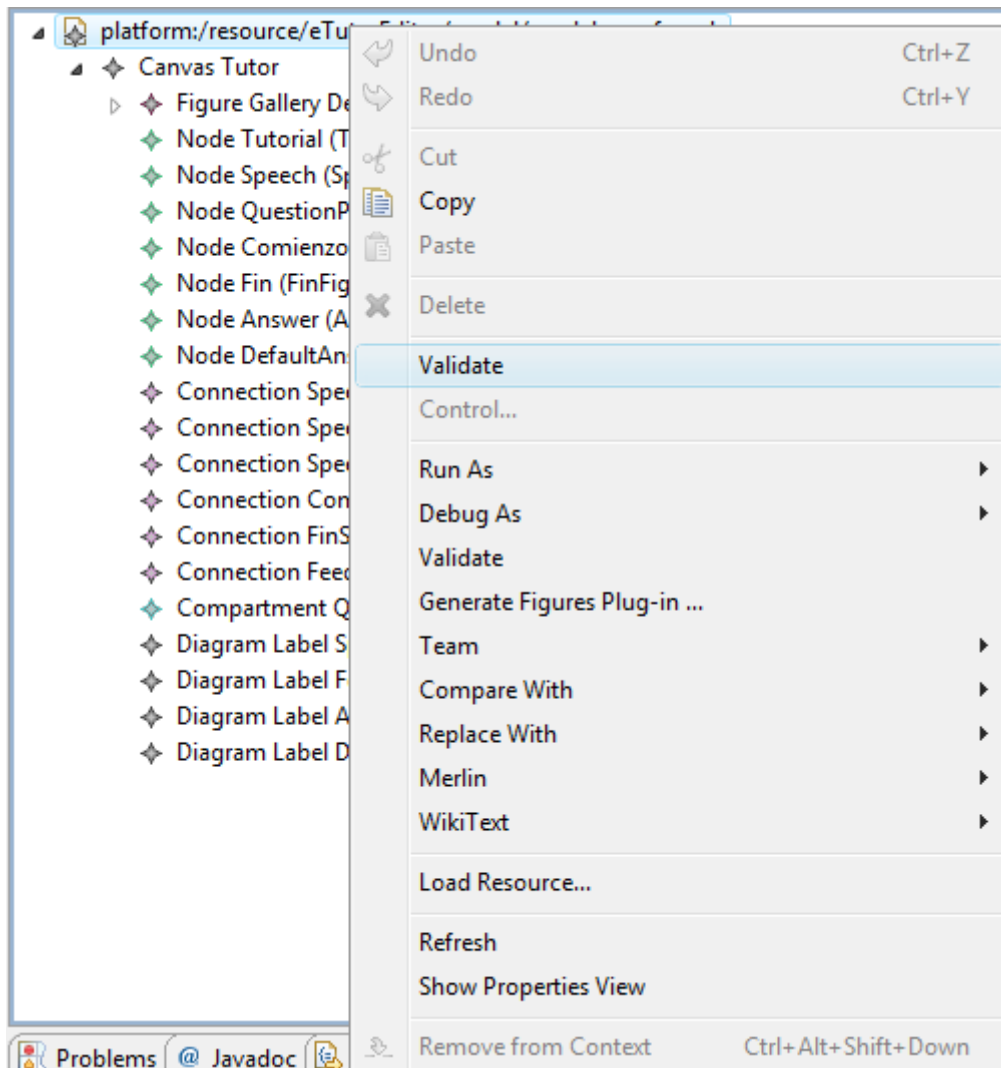
Compartment QPCompartment (QuestionPointFigure)

Property	Value
Accessor	Child Access getFigureQPCompartmentFigure
Collapsible	false
Figure	Figure Descriptor QuestionPointFigure
Name	QPCompartment
Needs Title	false

Diagram Label SpeechTexto

Property	Value
Accessor	Child Access getFigureSpeechTextoFigure
Affixed Parent Side	NONE
Container	
Content Pane	
Element Icon	true
External	false
Figure	Figure Descriptor SpeechFigure
Name	SpeechTexto
Resize Constraint	NSEW

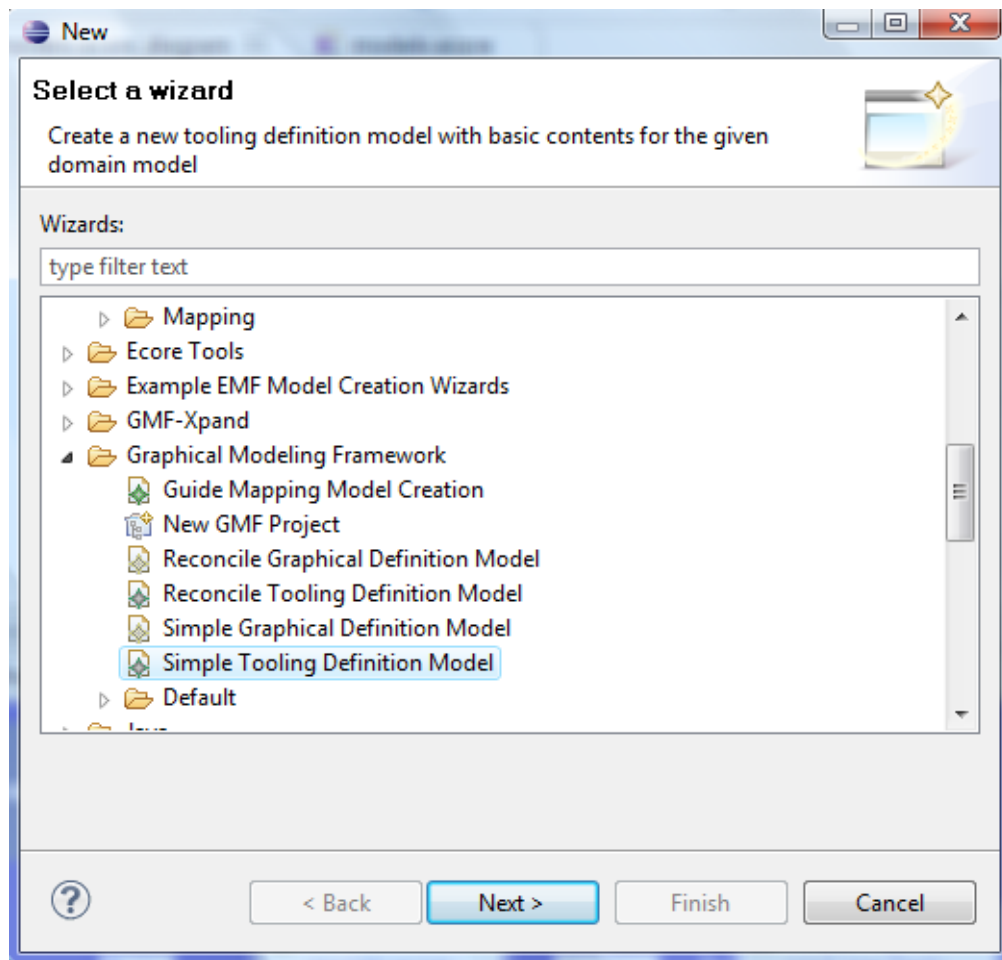
Una vez finalizado el proceso de definición gráfica, es conveniente validarlo para que Eclipse nos advierta de los errores que hemos cometido.



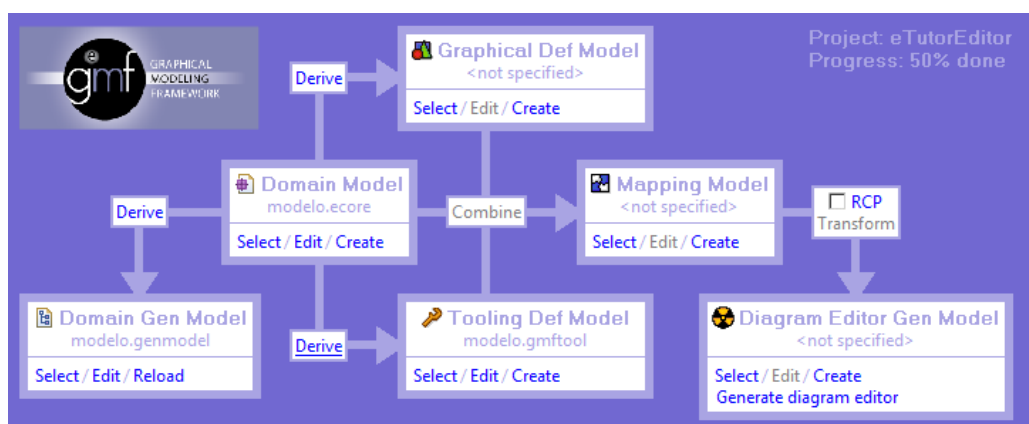
4.4.5.- Definición de la Paleta de Herramientas

En este apartado definiremos la paleta de la aplicación con la que construiremos los modelos, en nuestro caso los tutoriales. Para ello, tenemos dos opciones para crear el modelo.gmftool:

1. Haciendo click derecho sobre la carpeta del proyecto GMF, seleccionamos *File -> New -> Other*, nos dirigimos a la carpeta *Graphical Modeling Framework*, elegimos *Simple Tooling Definition Model* y pulsamos *Next*.

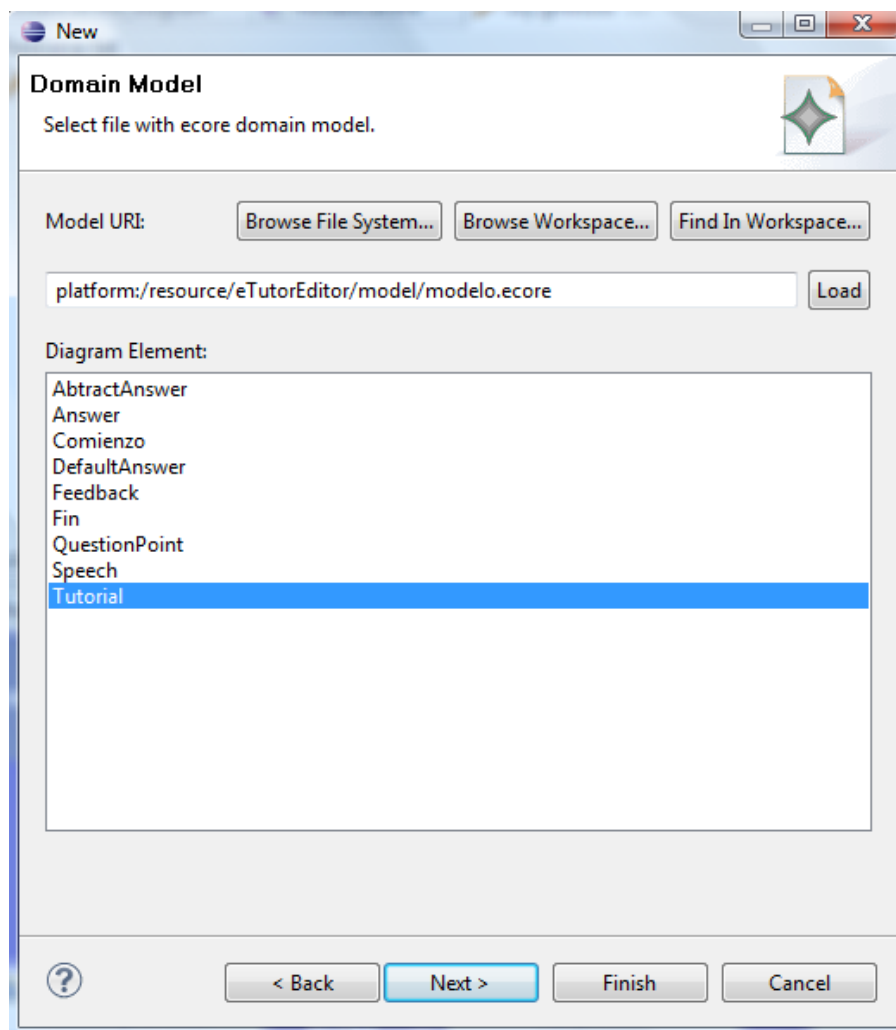


2. Utilizando el DashBoard, mediante el cual podremos derivar nuestro modelo gmftool a partir del modelo Ecore ya existente:



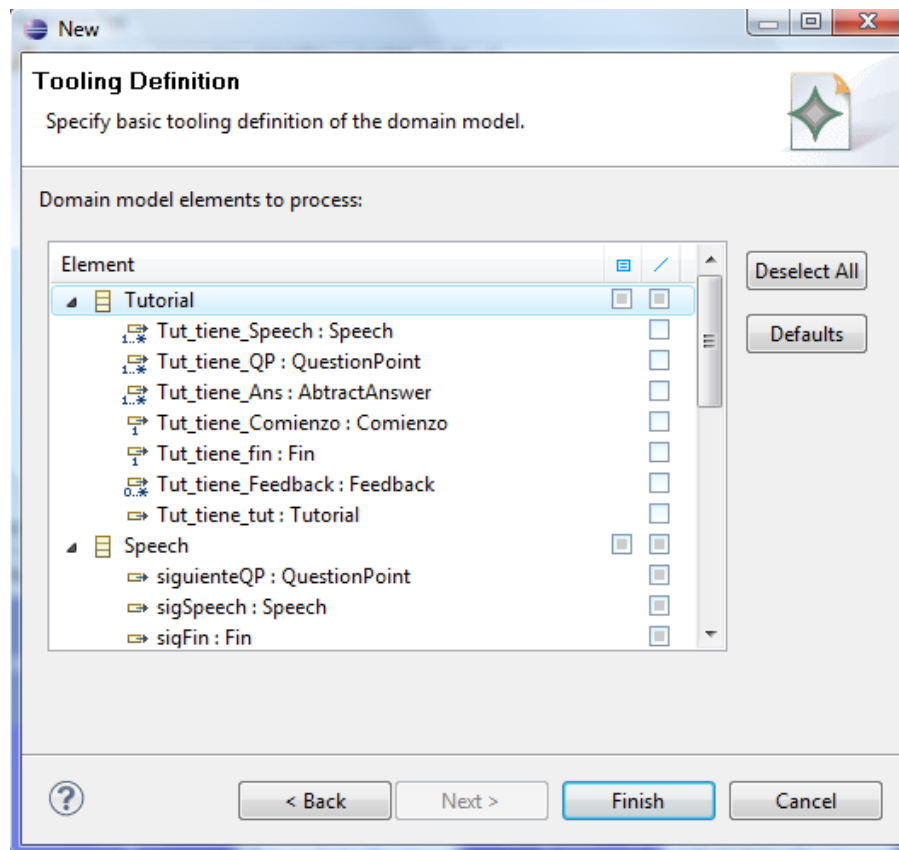
A continuación, damos un nombre a nuestro modelo de definición de herramientas y pulsamos *Next*.

Seleccionamos nuestro modelo .ecore como modelo de entrada. Si no aparece por defecto, pulsamos en *Find in Workspace* para seleccionarlo. Después, seleccionamos el elemento raíz de nuestro modelo, es decir, el que contiene al resto, en nuestro caso es Tutorial. Pulsamos *Next*.

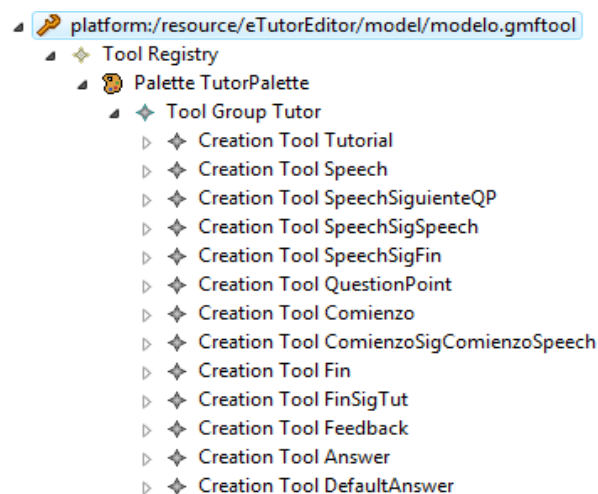


Al igual que en la definición gráfica, se nos presentan los distintos elementos del modelo y nos permite elegir aquellos que van aparecer como nodos o como enlaces en la barra de herramientas. Para ello se pueden seleccionar o deselectar los elementos que se deseen.

También podemos dejar los valores que la herramienta genera por defecto para después modificarlos manualmente.



Nos aparece un nuevo árbol, que representa la definición de herramientas de la aplicación a crear.



Ahora es el momento de modificar las características de nuestra paleta de herramientas, en el caso de que el generado no cumpliera con las especificaciones solicitadas.

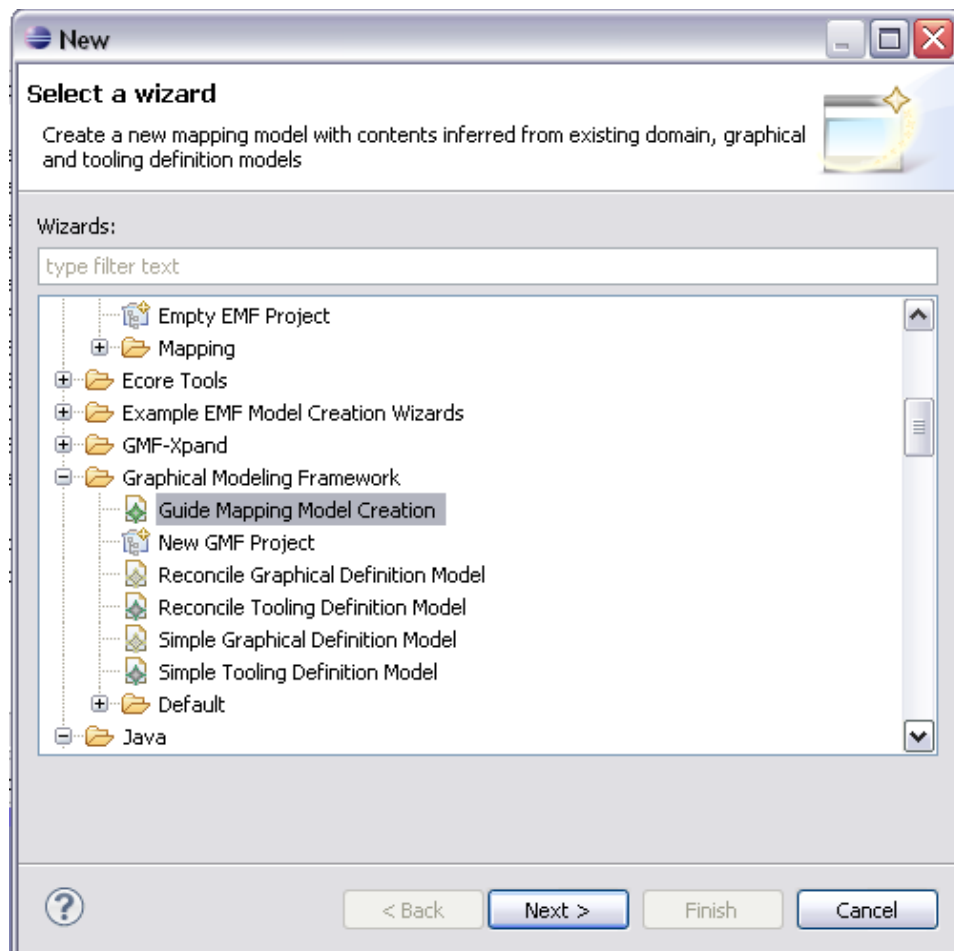
Al igual que hemos dicho para la definición gráfica, es conveniente validar por si hubiésemos cometido algún error a la hora de modificar la paleta de herramientas.

4.4.6.- Generación del mapeo de los elementos

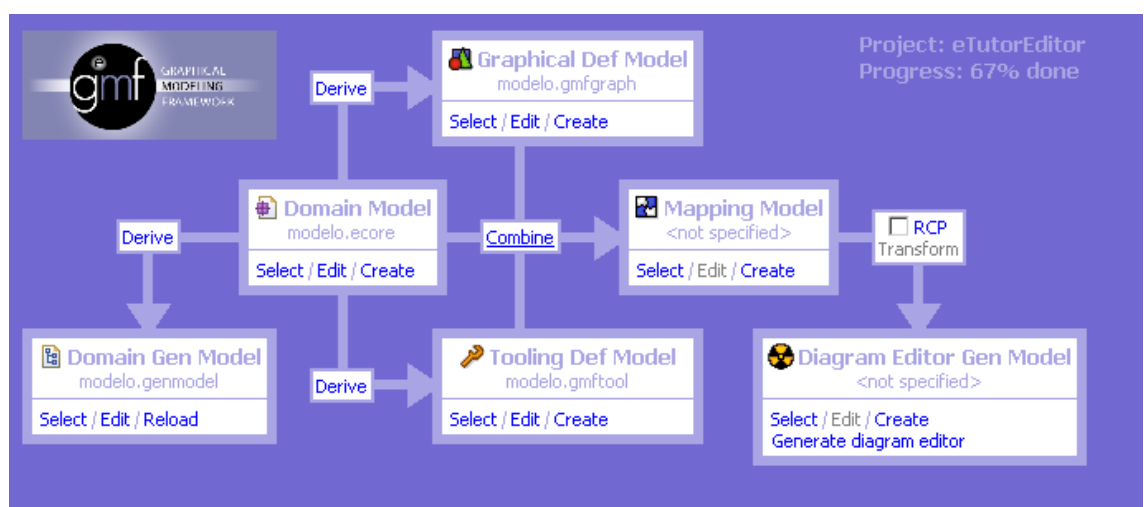
El proceso de mapping o correspondencia es aquél en el que todo lo creado anteriormente cobra un sentido y se une para formar la herramienta de construcción de modelos específicos de dominio. El mapeo se encargará de enlazar el modelo, la paleta y la definición gráfica, es decir, los tres modelos definidos anteriormente. Será en este paso donde conoceremos si toda la creación de esta herramienta tiene consistencia.

Debemos generar un archivo con extensión gmfmap, para lo cual podemos seguir dos métodos distintos:

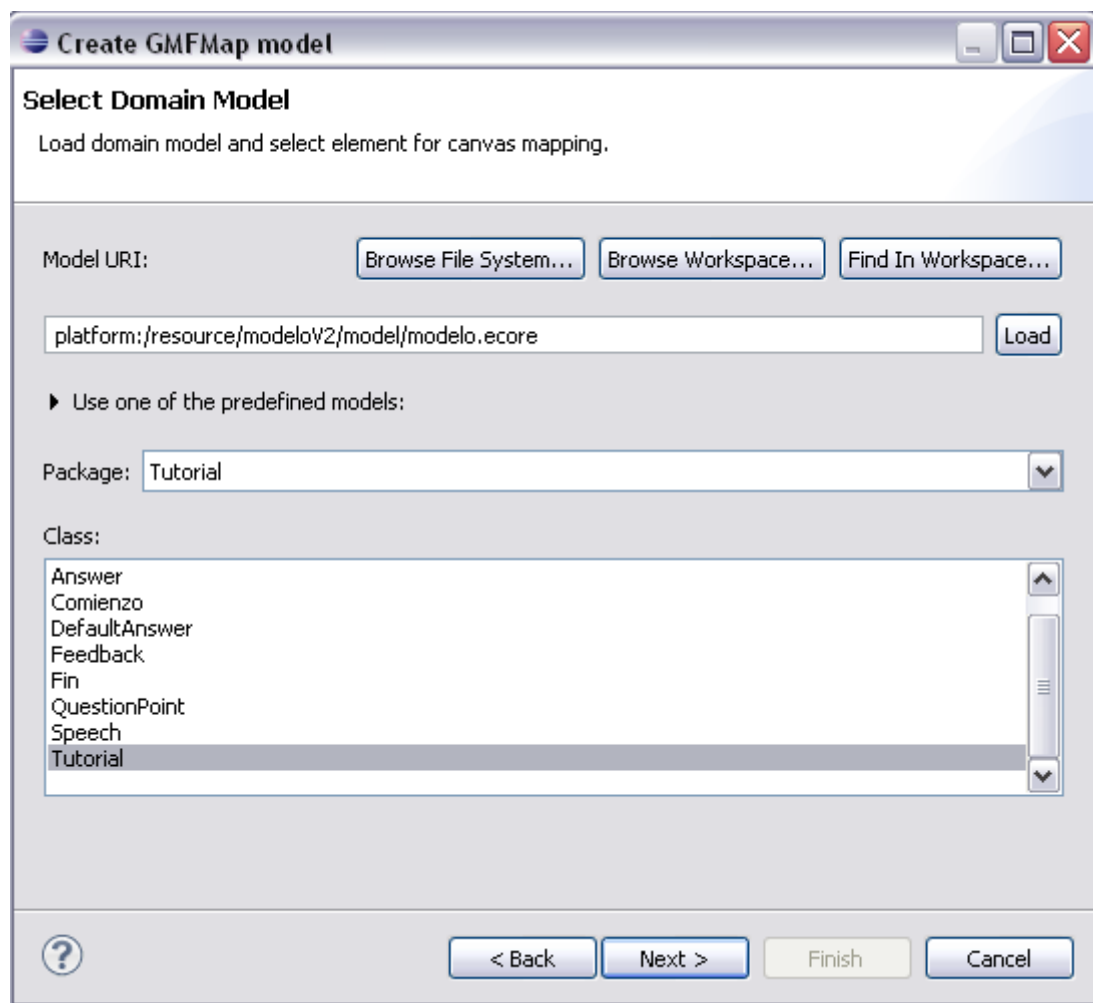
1. Hacemos click con el botón derecho sobre el proyecto GMF, utilizamos *New -> Other*, vamos a la carpeta *Graphical Modeling Framework* y seleccionamos *Guide Mapping Model Creation* y pulsamos *Next*.



2. Utilizamos el DashBoard, que permitirá crear este modelo combinando el modelo Ecore, el modelo Gmfggraph y el modelo Gmftool.

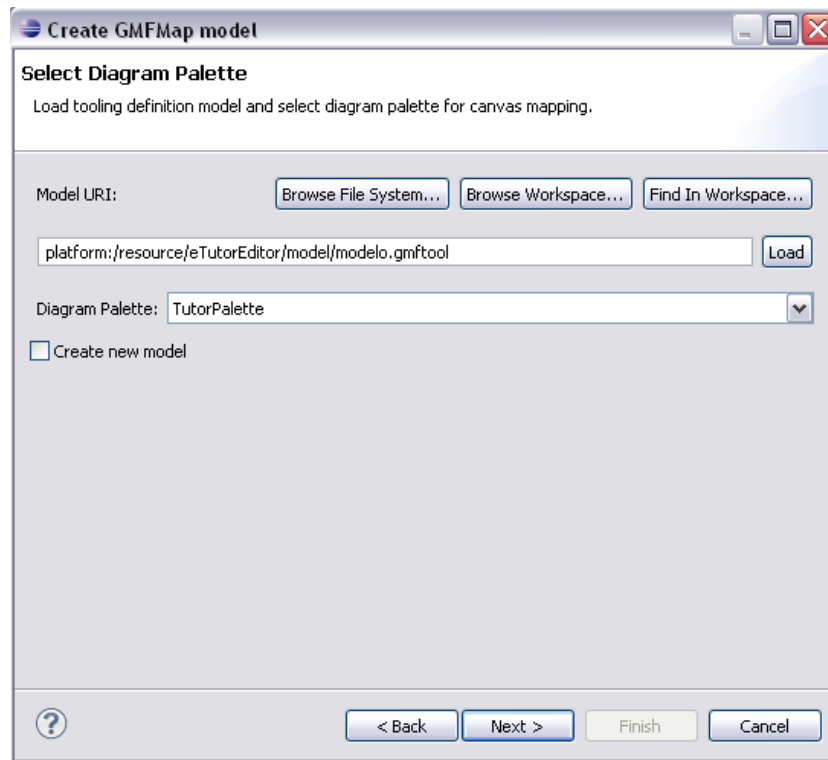


A continuación, seleccionamos la carpeta model de nuestro proyecto y le damos un nombre. En la siguiente ventana debemos relacionarlo con nuestro modelo Ecore.

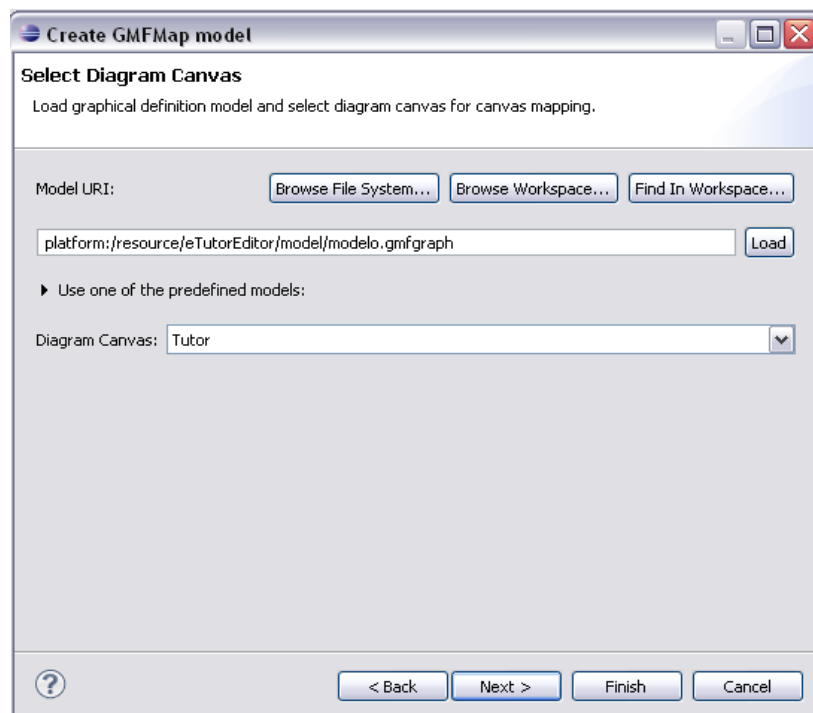


En la sección interior denominada Class, seleccionamos la clase principal que contiene a todas las demás, en nuestro caso será la clase Tutorial y pulsamos *Next*.

En la siguiente ventana, seleccionamos el modelo que contiene la configuración de la paleta que queremos utilizar. Pulsamos *Next*.

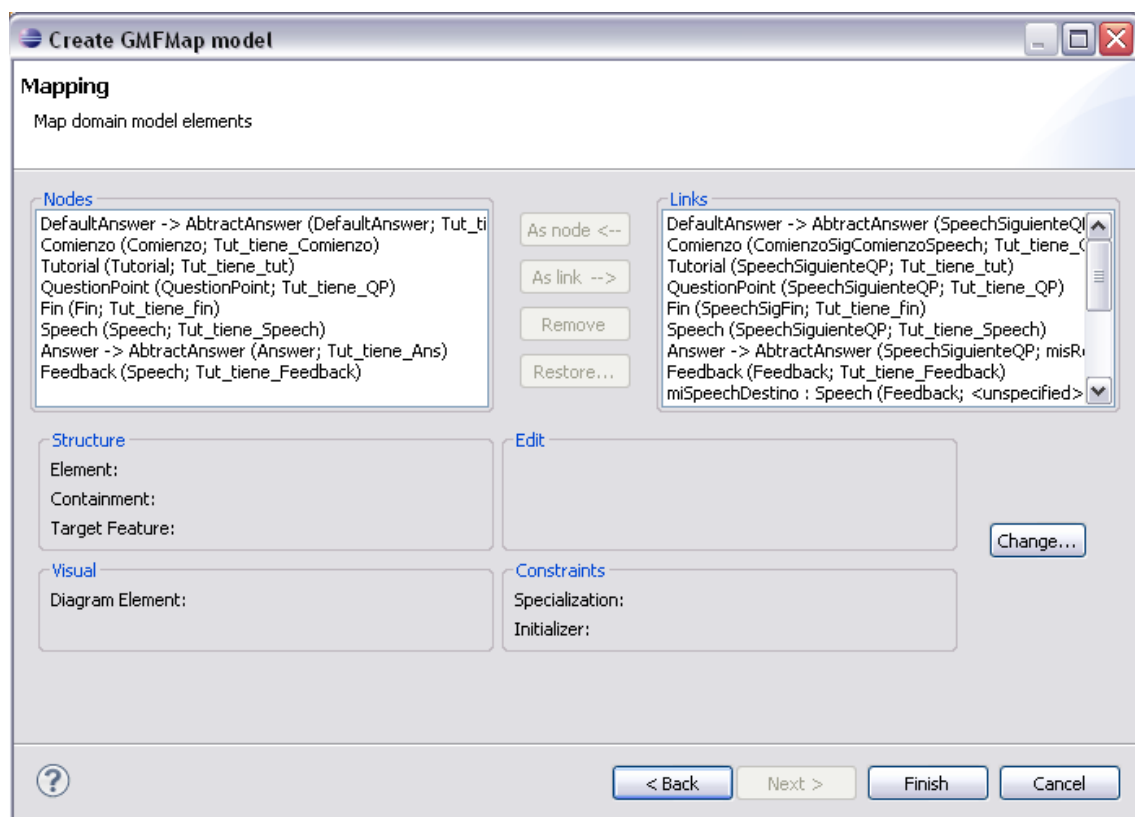


Ahora realizamos el mismo proceso para asignar el modelo gráfico que queremos asociar.



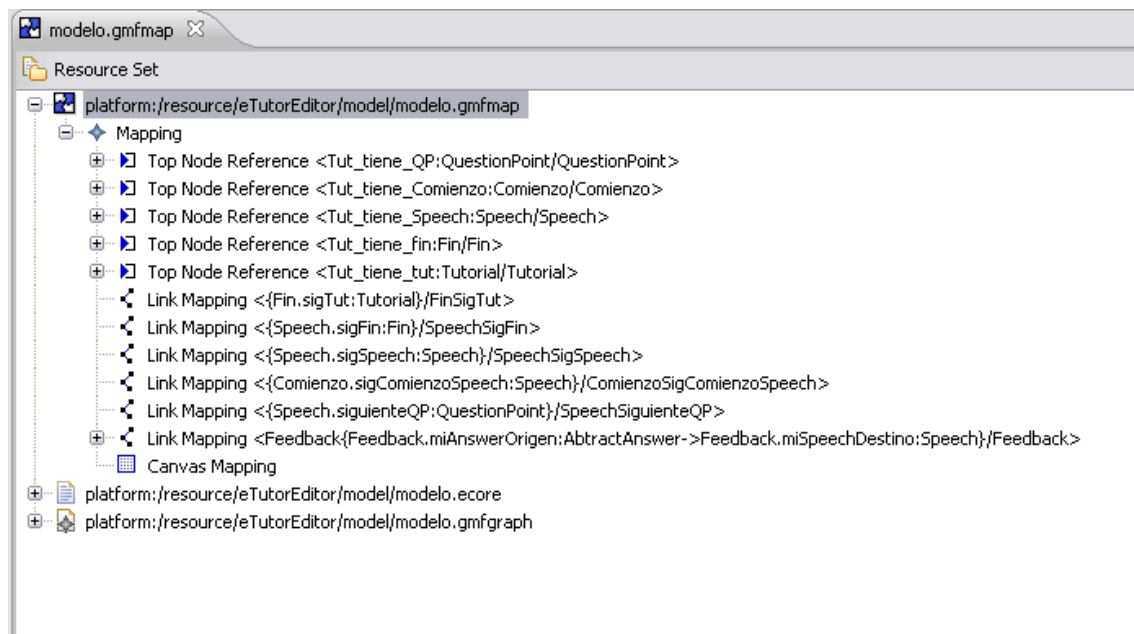
Si hemos creado anteriormente los modelos Gmfigraph y Gmftool en la misma carpeta de proyecto, al crear Gmfmap, nos aparecerán seleccionados por defecto.

A continuación, pasamos a la siguiente pantalla donde aparecen todos los elementos del modelo que se podrán dibujar o no en la paleta de la herramienta gráfica, para ello en esta pantalla se pueden seleccionar como nodos o enlaces los elementos que se deseen.



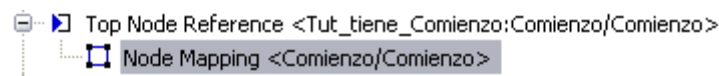
En este caso debemos asegurarnos que en la ventana *Nodes* aparecen los elementos que deseamos tener como nodos en nuestro diagrama, y la ventana *Links* los que queremos como conexiones. Los demás elementos los eliminaremos.

Una vez que hayamos pulsado *Finish* nos aparecerá el árbol de especificación del proceso de mapping.

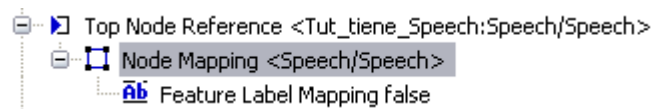


El fichero .gmfmap que se obtiene por defecto, no cumple las especificaciones solicitadas de nuestra herramienta gráfica, por lo que se ha tenido que transformar incorporando una serie de modificaciones y eliminando elementos que en nuestro caso no servían.

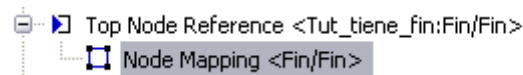
Tenemos que generar todos los elementos que queremos dibujar en la pizarra como nodos, y todos los enlaces entre elementos como link. En las propiedades de los nodos, hacemos corresponder el nodo del diagrama con su correspondiente icono en la paleta, ya que normalmente lo que aparece por defecto no concuerda. Es decir, al desplegar Top Node Reference de un elemento cualquiera y mostrar con Show Properties View las propiedades de Node Mapping debemos asociar cada nodo tanto con su herramienta de la paleta, como con su elemento gráfico.



Property	Value
Domain meta information	
Element	Comienzo
Misc	
Related Diagrams	
Visual representation	
Appearance Style	
Context Menu	
Diagram Node	Node Comienzo (ComienzoFigure)
Tool	Creation Tool Comienzo

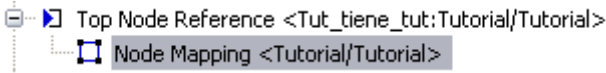


Property	Value
Domain meta information	
Element	Speech
Misc	
Related Diagrams	
Visual representation	
Appearance Style	
Context Menu	
Diagram Node	Node Speech (SpeechFigure)
Tool	Creation Tool Speech



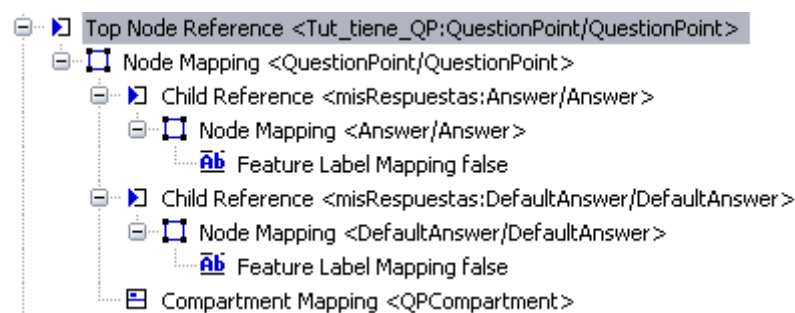
Property	Value
Domain meta information	
Element	Fin
Misc	
Related Diagrams	
Visual representation	
Appearance Style	
Context Menu	
Diagram Node	Node Fin (FinFigure)
Tool	Creation Tool Fin

En el nodo Tutorial además hay que darle valor al atributo Related Diagrams. Le asignamos Canvas Mapping lo que nos permitirá abrir un nuevo editor gráfico al hacer doble click sobre esta figura.



Property	Value
Domain meta information	
Element	Tutorial
Misc	
Related Diagrams	Canvas Mapping
Visual representation	
Appearance Style	
Context Menu	
Diagram Node	Node Tutorial (TutorialFigure)
Tool	Creation Tool Tutorial

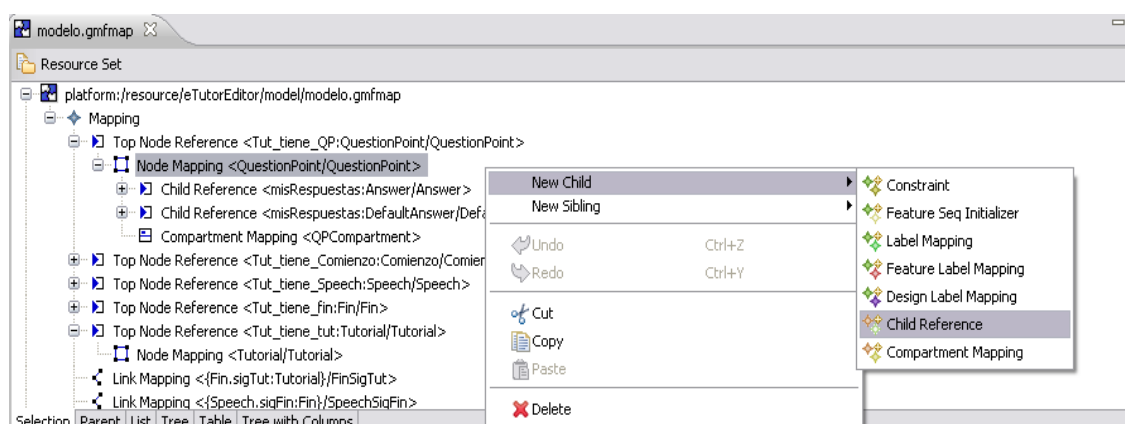
Un caso especial es el QuestionPoint, debido a que las Answer se dibujan en su interior. Por ello, debemos definirlos como hijas del QuestionPoint.



El nodo del QuestionPoint lo configuramos como Compartment. Para ello clicamos con el botón derecho sobre Node Mapping y seleccionamos *New Child -> Compartment Mapping*. Dentro de las propiedades, asociamos al Compartment su representación visual creada anteriormente en el modelo gráfico.

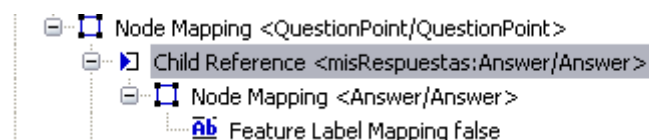
<div> <div>Compartment Mapping <QPCompartment></div> </div>	
Property	Value
Misc	
Children	Child Reference <misRespuestas:Answer/Answer>, Child Reference <misRespuestas:DefaultAnswer/DefaultAnswer>
Visual representation	
Compartment	Compartment QPCompartment (QuestionPointFigure)

Ahora, para crear las respuestas como hijas, hacemos click derecho sobre Node Mapping y seleccionamos *New Child -> Child Reference*.



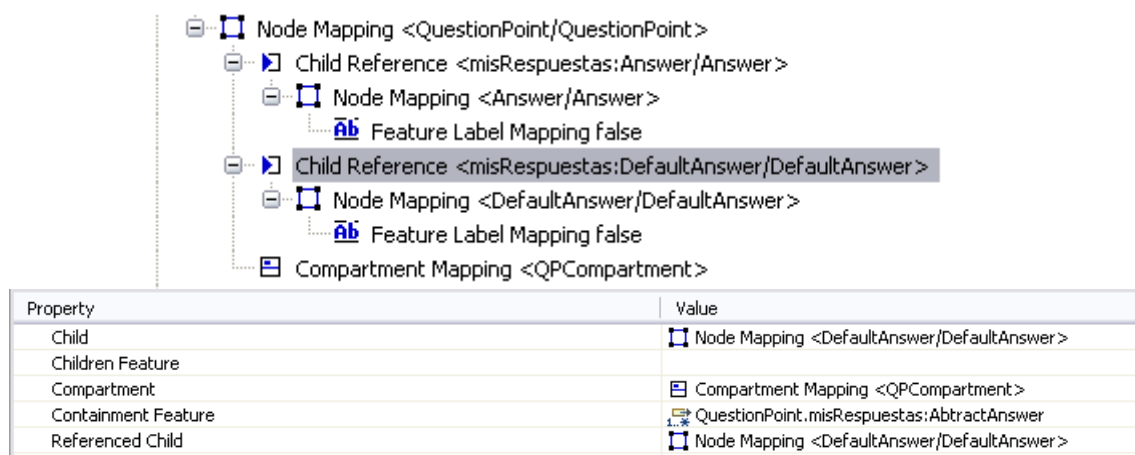
Necesitamos dos hijos diferentes, uno para cada tipo de respuesta. Dentro de cada Child Reference, debemos crear un nuevo Node Mapping, en el que asociamos el nodo con su representación en la paleta. Una vez hecho esto, tenemos que darle valores a los atributos de Child Reference tal y como se muestra en la siguientes figuras.

Configuración para Answer:

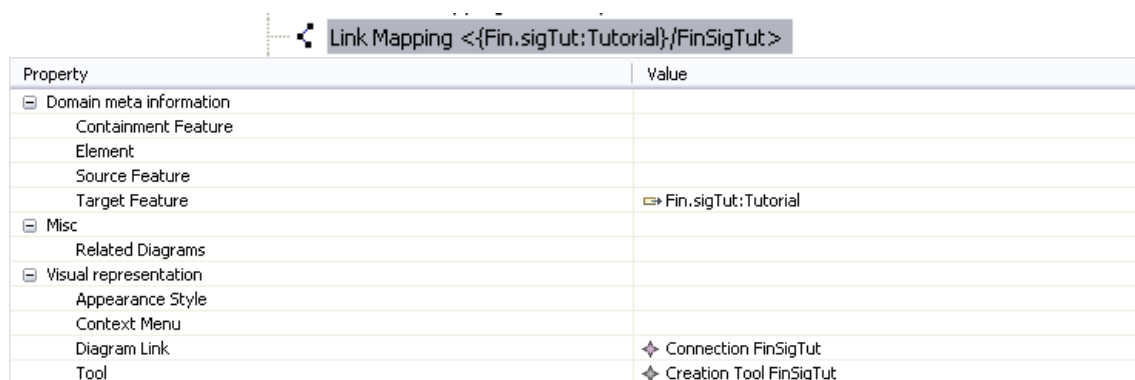


Property	Value
Child	Node Mapping <Answer/Answer>
Children Feature	
Compartment	Compartment Mapping <QPCompartment>
Containment Feature	QuestionPoint.misRespuestas:AbstractAnswer
Referenced Child	Node Mapping <Answer/Answer>

Configuración para DefaultAnswer:




A continuación, hay que comprobar que cada link está asociado con su herramienta de creación y su componente gráfica.





Link Mapping <{Speech.sigFin:Fin}/SpeechSigFin>	
Property	Value
[-] Domain meta information	
Containment Feature	
Element	
Source Feature	
Target Feature	⇒ Speech.sigFin:Fin
[-] Misc	
Related Diagrams	
[-] Visual representation	
Appearance Style	
Context Menu	
Diagram Link	◆ Connection SpeechSigFin
Tool	◆ Creation Tool SpeechSigFin

Link Mapping <{Speech.sigSpeech:Speech}/SpeechSigSpeech>	
Property	Value
[-] Domain meta information	
Containment Feature	
Element	
Source Feature	
Target Feature	⇒ Speech.sigSpeech:Speech
[-] Misc	
Related Diagrams	
[-] Visual representation	
Appearance Style	
Context Menu	
Diagram Link	◆ Connection SpeechSigSpeech
Tool	◆ Creation Tool SpeechSigSpeech

Link Mapping <{Comienzo.sigComienzoSpeech:Speech}/ComienzoSigComienzoSpeech>	
Property	Value
[-] Domain meta information	
Containment Feature	
Element	
Source Feature	
Target Feature	⇒ Comienzo.sigComienzoSpeech:Speech
[-] Misc	
Related Diagrams	
[-] Visual representation	
Appearance Style	
Context Menu	
Diagram Link	◆ Connection ComienzoSigComienzoSpeech
Tool	◆ Creation Tool ComienzoSigComienzoSpeech

 Link Mapping <{Speech.siguienteQP:QuestionPoint}/SpeechSiguienteQP>	
Property	Value
Domain meta information	
Containment Feature	
Element	
Source Feature	
Target Feature	Speech.siguienteQP:QuestionPoint
Misc	
Related Diagrams	
Visual representation	
Appearance Style	
Context Menu	
Diagram Link	Connection SpeechSiguienteQP
Tool	Creation Tool SpeechSiguienteQP

El caso de la conexión Feedback también es especial. Debemos crear un *Link Mapping* para ella, en caso de que no apareciese ya creado. Después tendremos que configurar, además, otras opciones. La propiedad Containment feature representa lo mismo que en un nodo. Debemos asociarle en la propiedad Element al elemento del modelo que representa. Las propiedades Source Feature y Target Feature apuntan a elementos definidos en el modelo de dominio. Estas indican de nuevo qué elementos serán origen y destino en una asociación, en nuestro caso son Answer y Speech, lo que indica que esta relación sólo puede establecerse entre estos dos elementos y con el sentido indicado, de Answer a Speech. Además, añadiremos un *Feature Label Mapping* que representa al label del Feedback.

 Link Mapping <Feedback{Feedback.miAnswerOrigen:AbtractAnswer->Feedback.miSpeechDestino:Speech}/Feedback>	
 Feature Label Mapping false	
Property	Value
Domain meta information	
Containment Feature	Tutorial.Tut_tiene_Feedback:Feedback
Element	Feedback
Source Feature	Feedback.miAnswerOrigen:AbtractAnswer
Target Feature	Feedback.miSpeechDestino:Speech
Misc	
Related Diagrams	
Visual representation	
Appearance Style	
Context Menu	
Diagram Link	Connection Feedback
Tool	Creation Tool Feedback

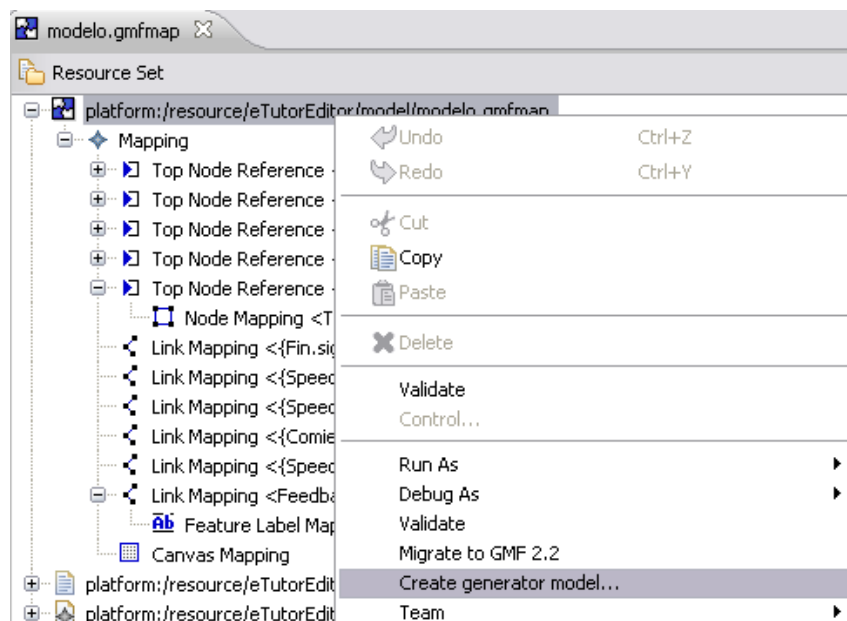
Tras este paso debemos comprobar que el mapping se ha realizado correctamente. Para ello, se ha de pinchar sobre Mapping con el botón derecho y seleccionar *Validate*.

4.4.6.- Generación de código

Una vez acabado el mapping, ya está el proceso de definición de nuestra herramienta completado. Ahora solamente falta generar su código, para que pueda ser ejecutada. Para generar el código, debemos tener un generador. Este modelo es creado por GMF a partir del modelo de mapeo y sólo se modifica para personalizar las opciones de generación del editor visual. Contiene toda la información de los cuatro modelos anteriores, y, la información necesaria para generar el código del editor visual.

Como en los modelos anteriores existen dos formas de crearlo:

1. Para conseguirlo debemos clicar con el botón derecho sobre la raíz del árbol de mapping y seleccionar *Create Generator Model*.



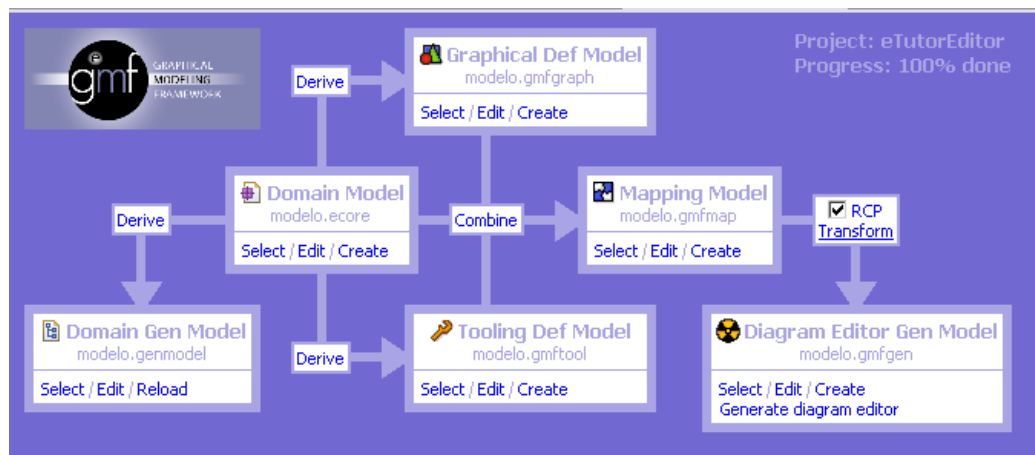
A continuación elegimos la carpeta destino model de nuestro proyecto GMF, le damos un nombre y pulsamos *Next*.

En las dos siguiente pantallas elegimos el modelo de mapeo, el modelo generador recientemente creado y pulsamos *Finish*.

De esta forma, se nos creará un nuevo archivo con extensión gmfgen.

Finalmente, pulsamos con el botón derecho sobre este nuevo archivo y seleccionamos *Generate Diagram Code*. Si se ha generado correctamente aparecerá un mensaje de confirmación. Se genera una carpeta con extensión diagram.

2. Utilizamos el DashBoard, que permitirá crear este modelo automáticamente pulsando Transform. Una vez creado el archivo generamos el código, pinchando en *Generate Diagram editor*, que aparecerá dentro de una nueva carpeta de extensión diagram.



Estos son todos los modelos que se requieren para crear el editor visual en GMF. Una vez generado el código, sólo queda ejecutar la aplicación como un plugin de Eclipse y verificar su funcionamiento.

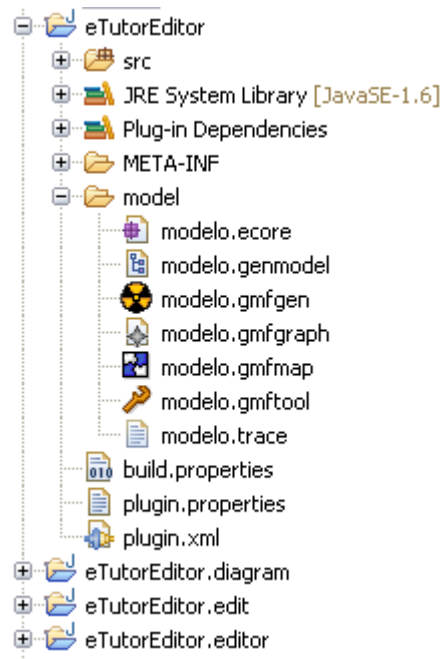
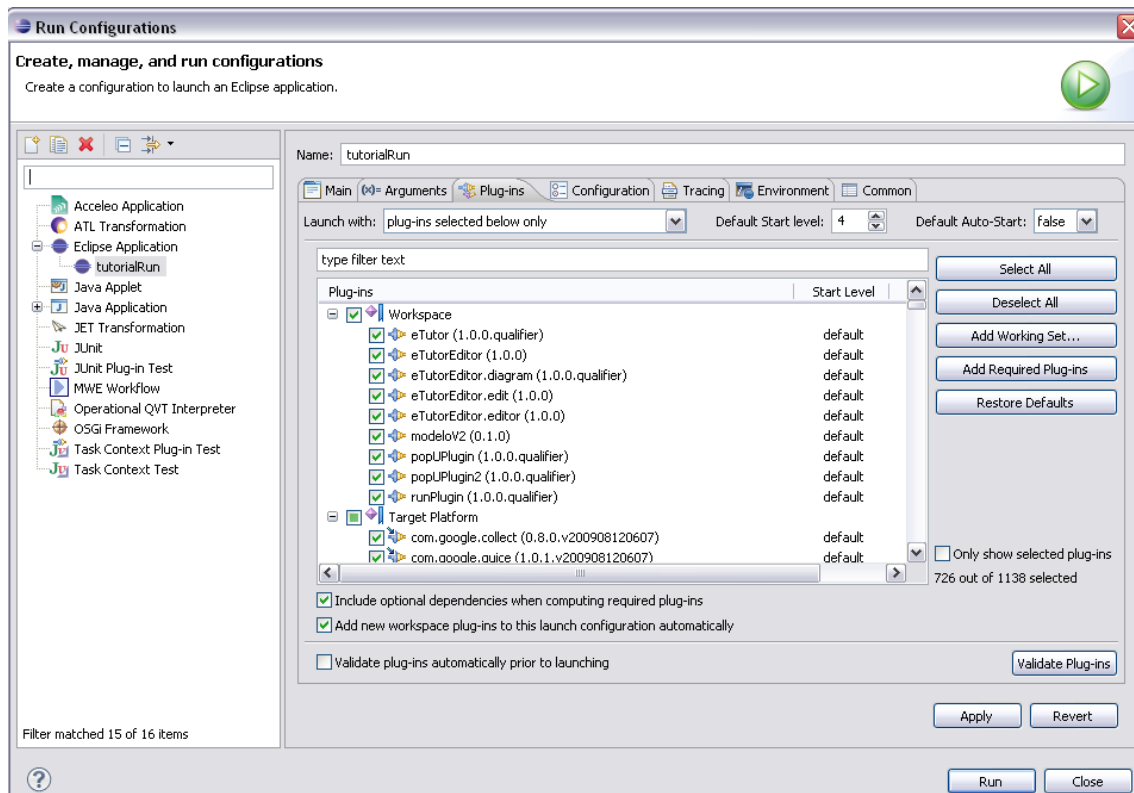


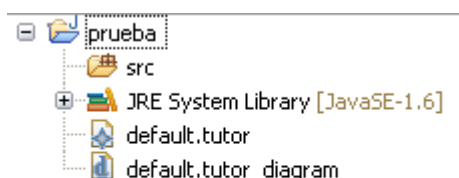
Figura 4.5 Estructura final del proyecto

4.4.7.- Ejecución de la herramienta de creación de diagramas

En este punto, ya disponemos de nuestra aplicación. Para ejecutarla, hay que seleccionar la carpeta con extensión diagram y pulsar sobre ejecutar. Seleccionar *Run as ->Run Configurations*. Se nos abrirá una nueva ventana en la que le damos un nombre a la ejecución. Después, en la pestaña de Plugins, debemos seleccionar los que se encuentran dentro de nuestro workspace. Pulsamos *Apply* y después *Run*.

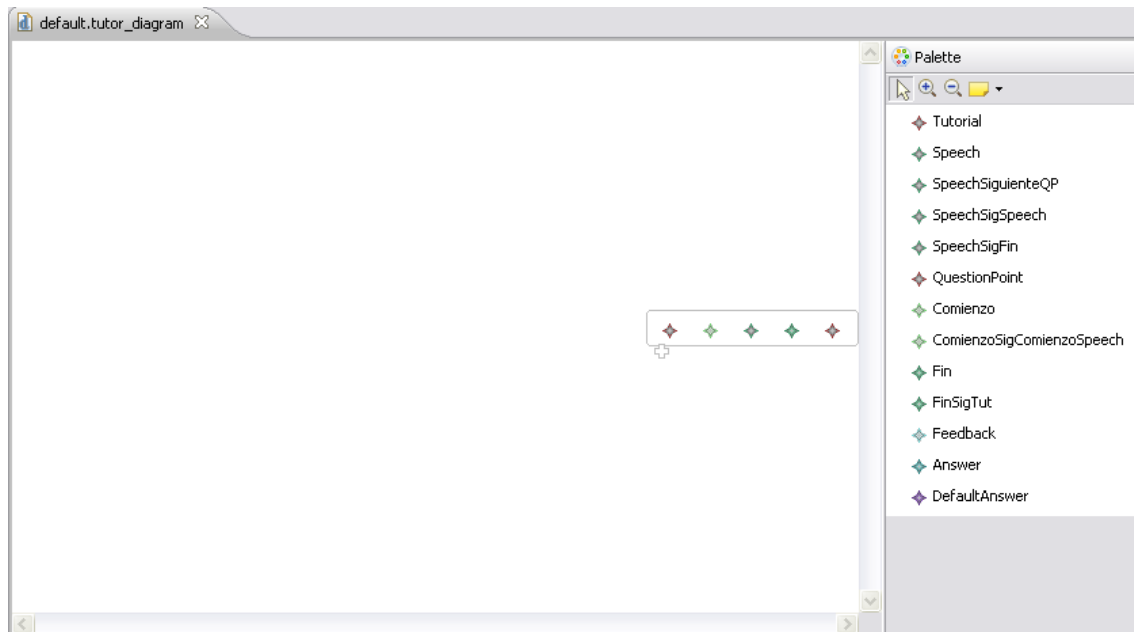


Se nos abrirá una nueva ventana de Eclipse. Creamos un nuevo proyecto java con *File -> New -> Java Project*. Le damos un nombre y pulsamos *Finish*. Dentro de este proyecto creamos un editor haciendo click derecho sobre el proyecto Java y seleccionamos *New -> Example -> Modelo Diagram*. Nos aparecerán dos archivos con extensiones tutor y tutor_diagram.

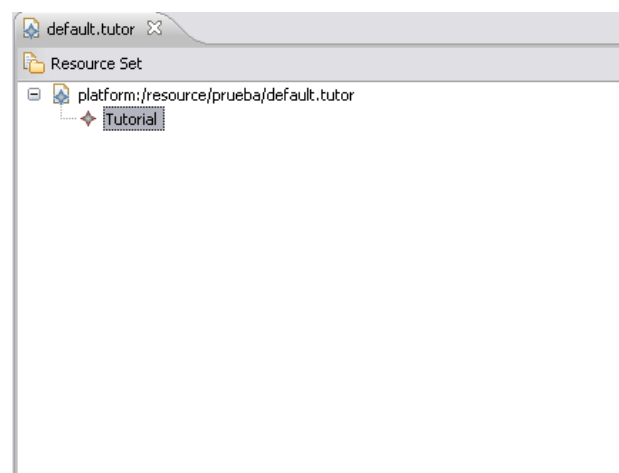


Al hacer doble click sobre el archivo de extensión tutor_diagram, se nos abrirá una pestaña que contiene en el centro una ventana de

edición en blanco y la paleta con todos los elementos que constituyen el tutorial a la derecha.



Si clicamos sobre el archivo con extensión tutor, se nos abrirá una pestaña que muestra el contenido del otro archivo, es decir, todos los elementos que contiene el tutorial, en forma de árbol. Inicialmente sólo contiene tutorial, que constituye el elemento raíz.

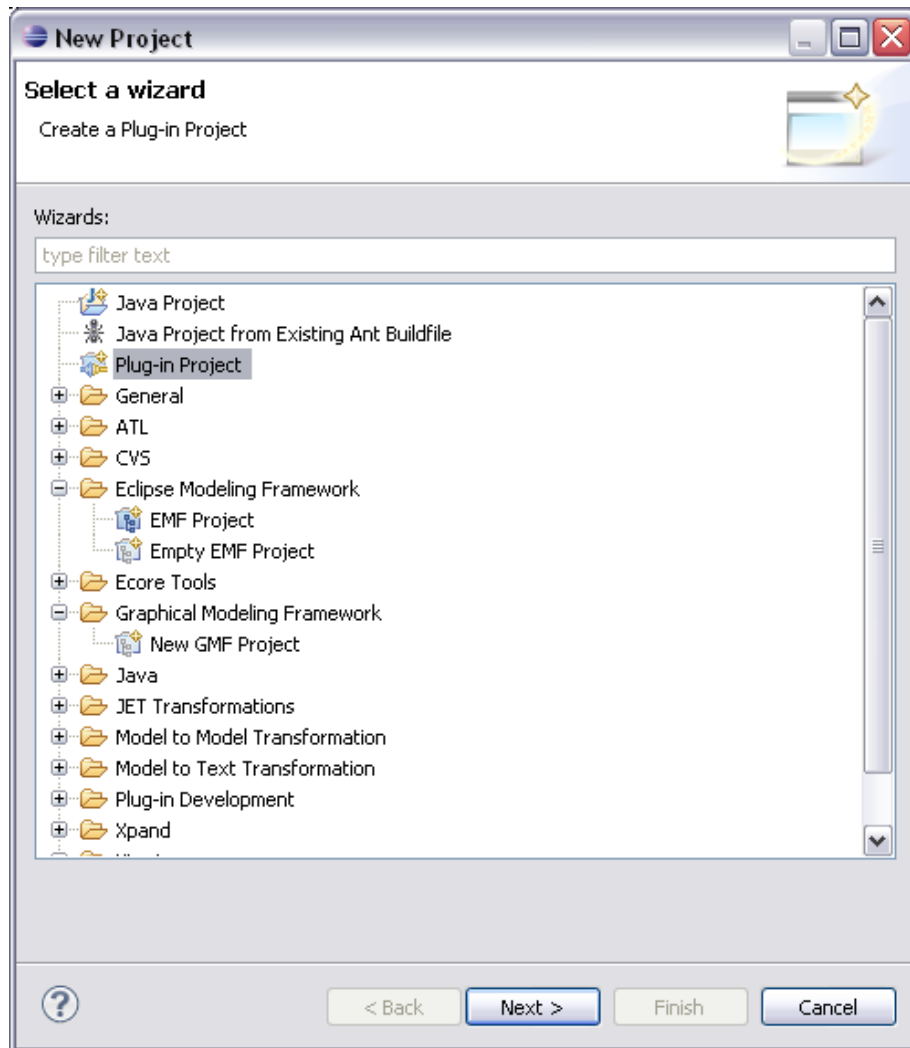


4.5.- Plugins <e-Tutor>GE

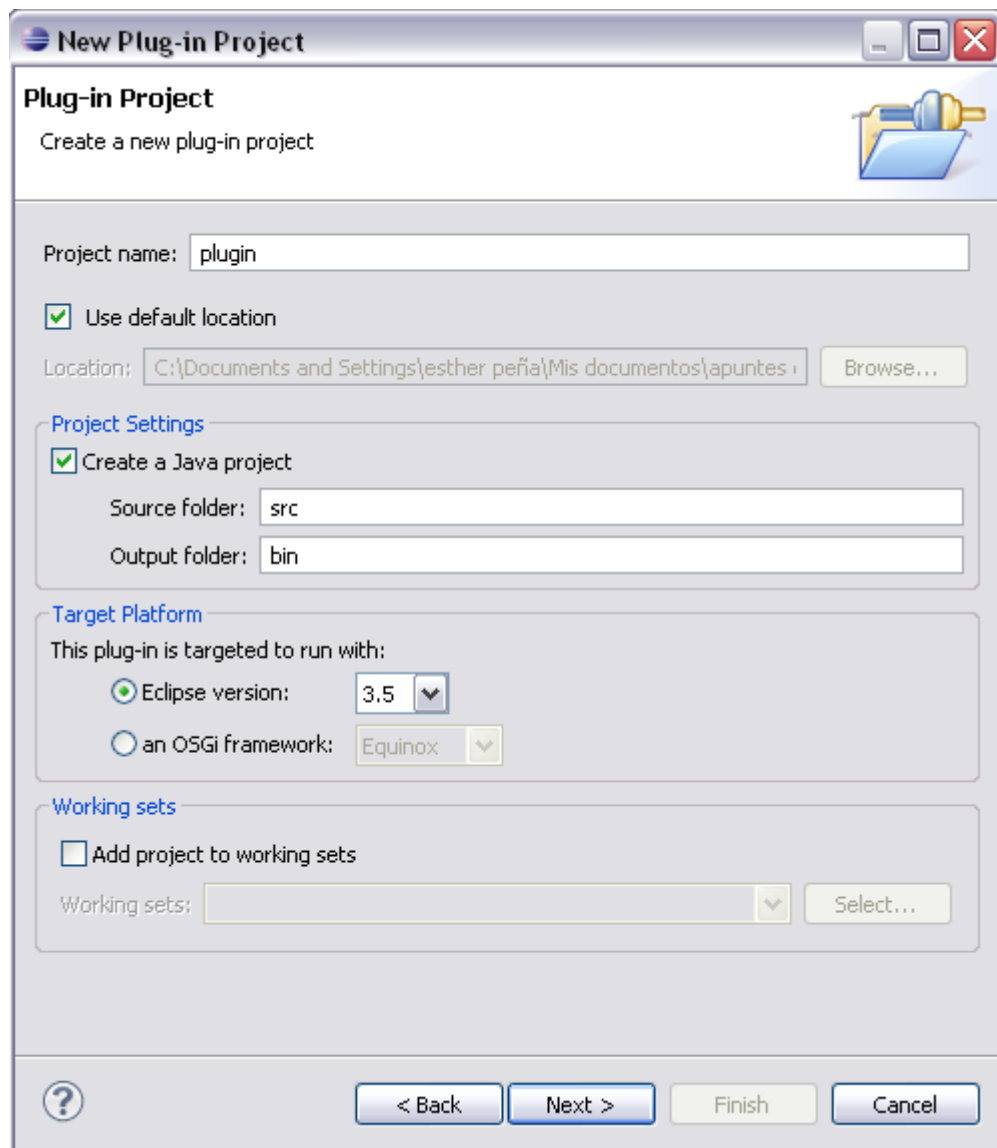
Este tipo de proyectos se usan para añadir nuevos módulos y funciones al entorno Eclipse. En nuestro caso, creamos plug-ins que nos permitan añadir un nuevo elemento al popUp Menu (Menú que aparece al hacer click derecho sobre una figura) con una funcionalidad asociada. En concreto creamos tres plug-ins: uno para cargar un archivo de texto (popUpPlugin1), otro para abrir una pestaña con un editor de texto y poder modificar el archivo (popUpPlugin2), y un tercero para ejecutar el tutorial (runPlugin). A continuación, se describen los tres pasos fundamentales a la hora de crear un proyecto de plug-in: la creación de proyecto, la modificación del archivo plugin.xml y MANIFEST.MF, y por ultimo, la implementación de las clases necesarias para añadir la funcionalidad deseada al plug-in.

4.5.1.- Creación de un proyecto Plugin

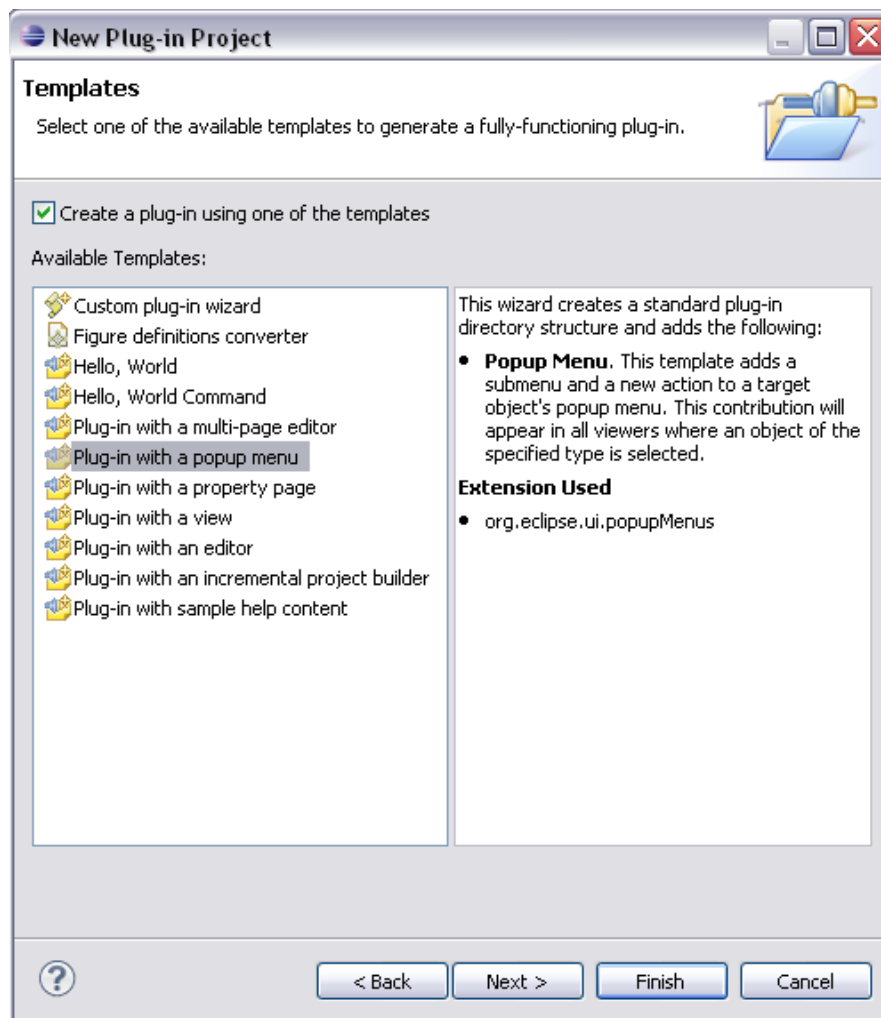
Para crear un nuevo proyecto vamos a *File -> New -> Project*, seleccionamos *Plug-in Project* y pasamos a la siguiente ventana.



Le proporcionamos un nombre al proyecto y pulsamos *Next*.



En la siguiente pantalla debemos desactivar la opción *Generate an activator* y pulsamos *Next*. A continuación, elegimos como plantilla *Plug-in with a popup menú* y terminamos pulsando *Finish*.



Como resultado se nos habrá creado un proyecto de plug-in como el que observamos en la Figura 4.6.

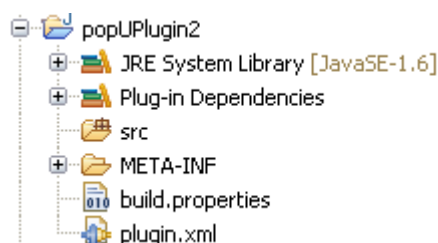
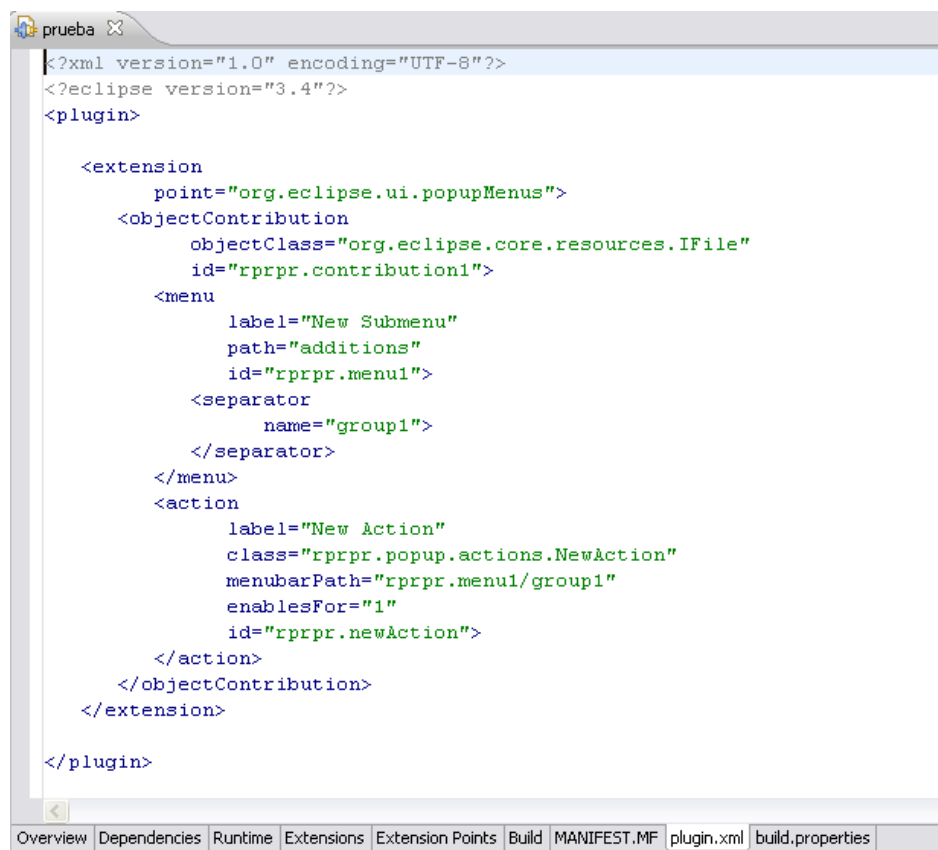


Figura 4.6 Estructura proyecto de plug-in

4.5.2.- Creación del archivo xml y modificación del MANIFEST

El archivo xml establece la relación entre el plug-in y el código Java que lo implementa. El archivo se denomina por defecto plugin.xml y se encuentra dentro del proyecto Plug-in. Para acceder al código pinchamos sobre plugin.xml y nos aseguramos de tener abierta la pestaña con el mismo nombre. Nos aparecerá una página con un código por defecto que podremos modificar.



```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>

    <extension
        point="org.eclipse.ui.popupMenus">
        <objectContribution
            objectClass="org.eclipse.core.resources.IFile"
            id="rprpr.contribution1">
            <menu
                label="New Submenu"
                path="additions"
                id="rprpr.menu1">
                <separator
                    name="group1">
                </separator>
            </menu>
            <action
                label="New Action"
                class="rprpr.popup.actions.NewAction"
                menubarPath="rprpr.menu1/group1"
                enablesFor="1"
                id="rprpr.newAction">
            </action>
        </objectContribution>
    </extension>

</plugin>
```

The screenshot shows the Eclipse IDE interface with a tab labeled 'prueba'. The main editor area displays the XML content of a plugin.xml file. The XML defines an extension for the 'org.eclipse.ui.popupMenus' point, which includes an object contribution for 'org.eclipse.core.resources.IFile' and a menu with a separator and an action. The bottom of the IDE shows a series of tabs: Overview, Dependencies, Runtime, Extensions, Extension Points, Build, MANIFEST.MF, plugin.xml (which is the active tab), and build.properties.

En nuestro caso borraremos el bloque menú, ya que no nos hace falta. A continuación, definimos dentro del bloque action:

- *Class*: la clase que contiene el código que se debe ejecutar.
- *Label*: el nombre que aparecerá en el menú emergente para realizar esta acción.

- *EnablesFor*: si es igual a uno, permanecerá activado, si su valor es cero, esta opción aparecerá desactivada en el popup menú.
- *Id*: identificador para la acción asociada al plugin.
- *MenubarPath*: para que la acción se muestre en el menú emergente inicial, sólo tenemos que especificar un nombre de grupo en este atributo. En este caso, utilizamos uno por defecto denominado additions.

También debemos definir los atributos del elemento `objectContribution`:

- *ObjectClass*: elemento en el que debe aparecer esta opción del menú emergente.
- *Id*: identificador único para este punto de extensión.

El resultado de nuestro código xml, para la definición del plugin editar archivo, será el mostrado en la Figura 4.7.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>

    <extension
        point="org.eclipse.ui.popupMenus">
        <objectContribution
            id="Tutorial.Tutorial.diagram.ui.objectContribution.SpeechEditPart1"
            objectClass="Tutor.Tutor.diagram.edit.parts.SpeechEditPart">
            <action
                class = "Tutor.Tutor.diagram.part.EditarArchivo"
                enablesFor="1"
                menubarPath="additions"
                id="Tutorial.Tutorial.diagram.part.EditarArchivoID"
                label="Edit Speech File">
            </action>
        </objectContribution>
    </extension>

</plugin>
```

Figura 4.7 Código xml del plug-in Editar Archivo

La Figura 4.8 muestra el código para el plugin cargarArchivo.

```

<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.5"?>
<plugin>
  <extension
    point="org.eclipse.ui.popupMenus">
    <objectContribution
      id="Tutorial.Tutorial.diagram.ui.objectContribution.SpeechEditPart1"
      objectClass="Tutor.Tutor.diagram.edit.parts.SpeechEditPart">
      <action
        class = "Tutor.Tutor.diagram.part.CargaArchivo"
        enablesFor="1"
        menubarPath="additions"
        id="Tutorial.Tutorial.diagram.part.CargaArchivoID"
        label="Load Speech File">
      </action>
    </objectContribution>
  </extension>
</plugin>

```

Figura 4.8 Código xml del plug-in Cargar Archivo

Para el plugin que ejecuta el tutorial, el código se muestra en la Figura 4.9.

```

<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>

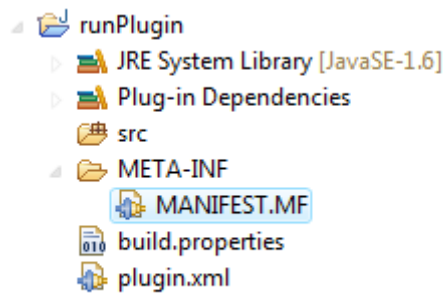
  <extension
    point="org.eclipse.ui.popupMenus">
    <objectContribution
      id="org.eclipse.ui.articles.action.contribution.popup.object"
      objectClass="Tutor.Tutor.diagram.edit.parts.TutorialEditPart">

      <action
        class="Tutor.Tutor.diagram.part.RunTutor"
        enablesFor="1"
        menubarPath="additions"
        id="Tutor.Tutor.diagram.part.RunTutorID"
        label="RunTutor">
      </action>
    </objectContribution>
  </extension>
</plugin>

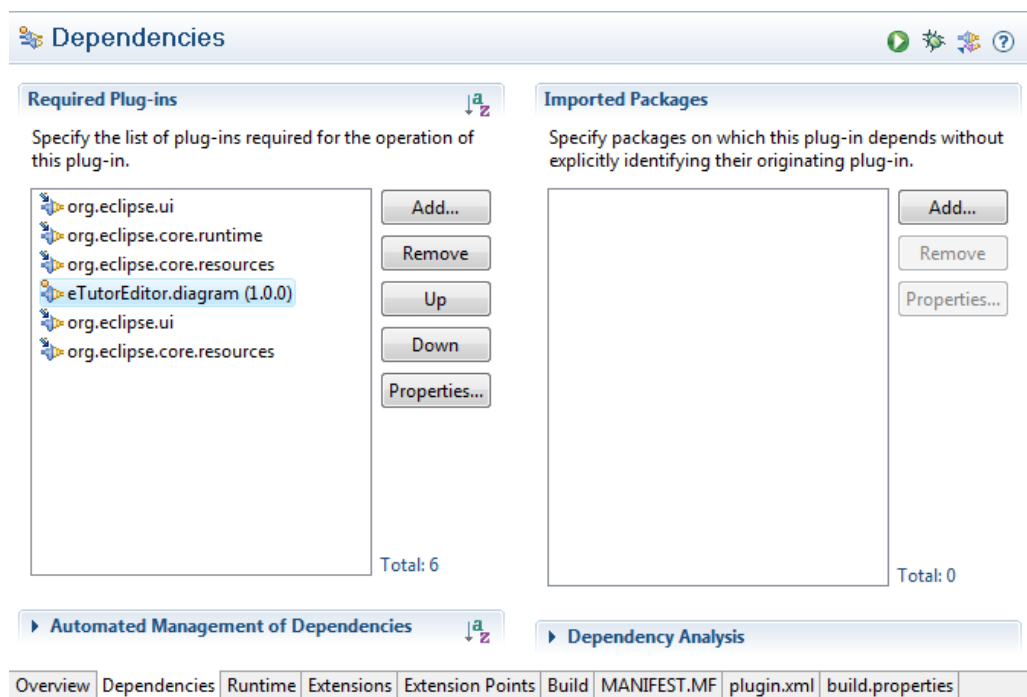
```

Figura 4.9 Código xml del plug-in RunTutor

Por otro lado, debemos hacer unas modificaciones dentro del archivo MANIFEST.MF. Este archivo se encuentra localizado dentro de la carpeta META-INF del proyecto de plug-in.



Abrimos este archivo y vamos a la pestaña *Dependencies*. Aquí, en la sección Required Plug-ins debemos añadir el proyecto eTutorEditor.diagram, puesto que las clases que implementarán las funciones de los plug-ins irán localizadas en este proyecto.



4.5.3.- Implementación del código

El código relacionado con el plug-in debe ser implementado en el proyecto GMF, explicado anteriormente. Concretamente se incluye en

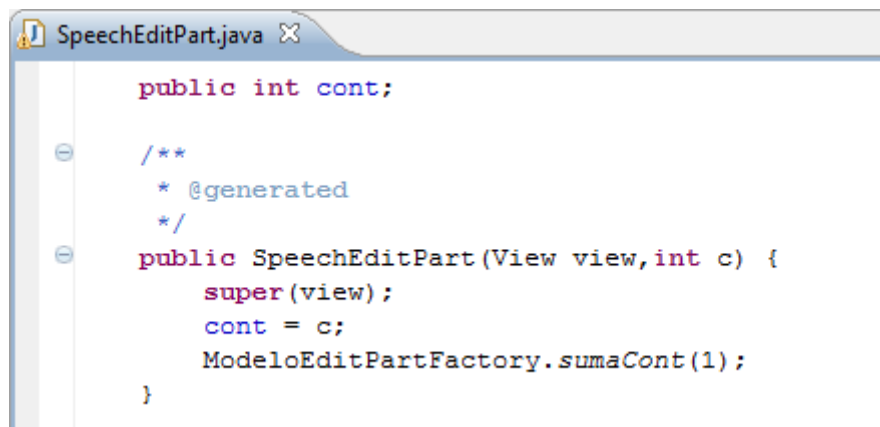
las clases que contienen los paquetes localizados en la carpeta src del proyecto con extensión .diagram.

- *PopUpPlugin1*: el primer paso antes de abordar la programación del código del plug-in es asociar a cada Speech un archivo. Para ello debemos modificar el código que GMF genera por defecto. Dentro del paquete Tutor.Tutor.diagram.edit.parts, cambiamos la clase que se corresponde con el EditPart del Speech, es decir, *SpeechEditPart.java*. En esta clase modificamos la constructora del propio Speech, añadiendo un atributo entero que será el número del archivo que llevará asociado. Además, ampliamos el método que se ejecuta cuando añadimos una nueva figura de este tipo al tutorial, *createNodeShape*. Aquí creamos el archivo, con un nombre por defecto, que se corresponderá con el atributo que introducimos antes. Además, en la factoría de Editparts, en la clase *ModeloEditPartFactory*, añadimos un atributo estático *contSpeech* para llevar un conteo del número de Speech creados y así poder asignarle un nombre por defecto al archivo asociado (archivo0, archivo1, archivo2...). Este contador se introduce como parámetro en la llamada a la constructora de *SpeechEditPart*. Por lo tanto deberemos modificar también esta constructora, añadiéndole el nuevo parámetro creado.

De esta forma, desde el momento en que incluimos un nuevo Speech en nuestro tutorial, tendrá un archivo de texto asociado. Tras estas modificaciones, añadimos una nueva clase dentro del paquete Tutor.Tutor.diagram.part denominada *CargaArchivo*, que es la que aportará la función deseada al plug-in. Ésta, implementa a la interfaz *IObjectActionDelegate* que a su vez extiende a *ActionDelegate*, una clase abstracta que nos permite

desarrollar el método *run* relacionado con la acción que ejecutará el plugin.

En este caso el método *run* mostrará al usuario un *FileChooser* para que elija el archivo que desea cargar. Una vez elegido correctamente el archivo, lo asociamos a la figura seleccionada, que en este caso siempre será un *Speech*. Para saber qué *Speech* está seleccionado recorreremos todas las *EditParts* de nuestro diagrama. Una vez encontrada aquélla que está seleccionada, cambiamos el archivo que teníamos asociado al *Speech* por otro diferente. Además añadimos el código necesario para que los cambios queden reflejados en el modelo y la información sea persistente.



```
SpeechEditPart.java X
public int cont;

/**
 * @generated
 */
public SpeechEditPart(View view, int c) {
    super(view);
    cont = c;
    ModeloEditPartFactory.sumaCont(1);
}
```

```
SpeechEditPart.java X

/**
 * @generated
 */
protected IFigure createNodeShape() {
    SpeechFigure figure = new SpeechFigure();
    IWorkspace w = ResourcesPlugin.getWorkspace();
    IPath p = w.getRoot().getLocation();
    File fich = new File(p.toOSString()+"\\archivo"+cont+".txt");
    if(!fich.exists()){
        try {
            fich.createNewFile();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    NodeImpl ni = (NodeImpl) this.getModel();
    SpeechImpl si = (SpeechImpl) ni.getElement();
    String ruta = new String(fich.getAbsolutePath());
    String s = si.getTexto();
    figure.getFigureSpeechTextToFigure().setText(s);
    if(s == null){
        si.setTexto(ruta);
    }
    return primaryShape = figure;
}
```

```
ModeloEditPartFactory.java X

public static int contSpeech;

public static void sumaCont(int i) {
    // TODO Auto-generated method stub
    contSpeech = contSpeech+i;
}

public EditPart createEditPart(EditPart context, Object model) {
    if (model instanceof View) {
        View view = (View) model;
        switch (Tutor.Tutor.diagram.part.ModeloVisualIDRegistry
            .getVisualID(view)) {

            case Tutor.Tutor.diagram.edit.parts.TutorialEditPart.VISUAL_ID:
                return new Tutor.Tutor.diagram.edit.parts.TutorialEditPart(view);

            case Tutor.Tutor.diagram.edit.parts.QuestionPointEditPart.VISUAL_ID:
                return new Tutor.Tutor.diagram.edit.parts.QuestionPointEditPart(
                    view);

            case Tutor.Tutor.diagram.edit.parts.ComienzoEditPart.VISUAL_ID:
                return new Tutor.Tutor.diagram.edit.parts.ComienzoEditPart(view);

            case Tutor.Tutor.diagram.edit.parts.SpeechEditPart.VISUAL_ID:
                return new Tutor.Tutor.diagram.edit.parts.SpeechEditPart(view, contSpeech);
        }
    }
}
```

```
CargaArchivo.java
@Override
public void run(IAction action) {
    // TODO Auto-generated method stub
    JFrame fr = new JFrame();
    JFileChooser fc = new JFileChooser();
    int modoCerrar = fc.showOpenDialog(fr);
    if(modoCerrar == JFileChooser.APPROVE_OPTION){
        File f = fc.getSelectedFile();
        DiagramEditor editor = (DiagramEditor) PlatformUI.getWorkbench().
            getActiveWorkbenchWindow().getActivePage().getActiveEditor();
        List<AbstractEditPart> ep = editor.getDiagramEditPart().getChildren();
        boolean encontrado = false;
        int i = 0;
        EditPart editP = null;
        while(i<ep.size() && !encontrado){
            editP = (EditPart) ep.get(i);
            int g = editP.getSelected();
            if (g == EditPart.SELECTED_PRIMARY){
                encontrado = true;
            }
            else{i++;}
        }
        SpeechFigure sf = ((SpeechEditPart) editP).getPrimaryShape();
        sf.getFigureSpeechTextToFigure().setText(f.getAbsolutePath());
        NodeImpl ni = (NodeImpl) editP.getModel();
        SpeechImpl si = (SpeechImpl) ni.getElement();
        String ruta = new String(f.getAbsolutePath());
        si.setTexto(ruta);
    }
}
```

- *PopUpPlugin2*: en este caso, incluimos en el paquete denominado Tutor.Tutor.diagram.part, una nueva clase llamada EditarArchivo, donde escribiremos el código del plug-in. Esta clase también implementa a la interfaz IObjectActionDelegate y extiende a ActionDelegate.

El método run debe encontrar el elemento concreto que hemos seleccionado, buscando entre todas las EditParts de nuestro diagrama. Una vez encontrada, obtiene la ruta del archivo asociado a la EditPart seleccionada (en nuestro caso siempre será un Speech). Además obtenemos también el espacio de trabajo donde se localiza nuestro diagrama, para poder abrir una nueva pestaña de editor de texto con el archivo que se encuentra en la ruta correspondiente.

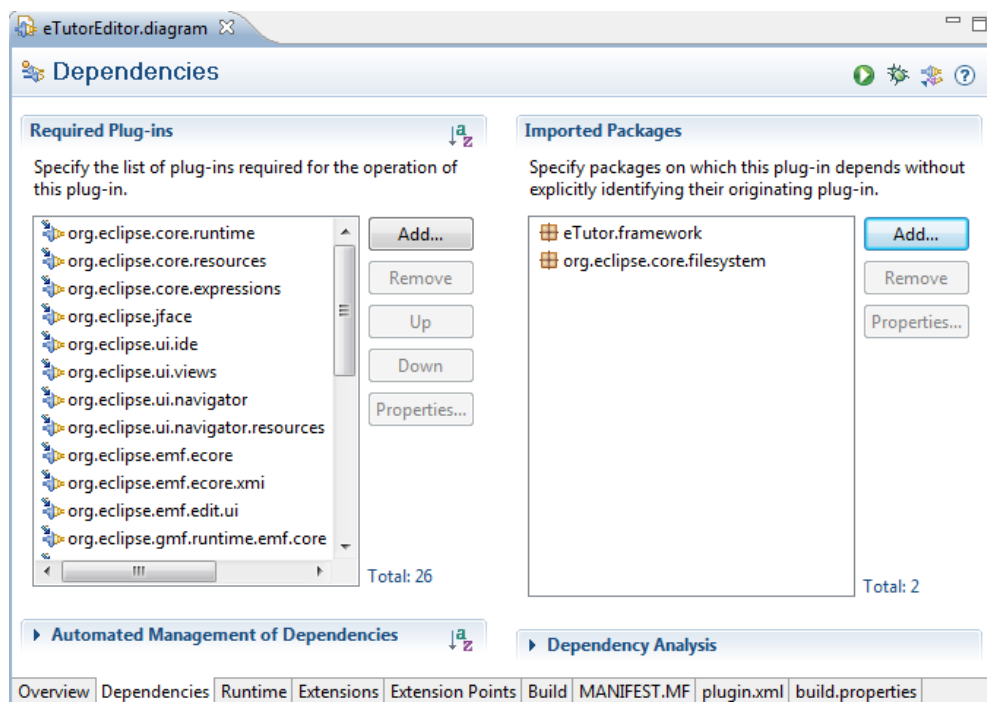

```
EditorArchivo.java X
public void run(IAction action) {
    DiagramEditor editor = (DiagramEditor) PlatformUI.getWorkbench().
        getActiveWorkbenchWindow().getActivePage().getActiveEditor();
    List<AbstractEditPart> ep = editor.getDiagramEditPart().getChildren();
    boolean encontrado = false;
    int i = 0;
    EditPart editP = null;
    while(i<=ep.size() && !encontrado){
        editP = (EditPart) ep.get(i);
        int g = editP.getSelected();
        if (g == EditPart.SELECTED_PRIMARY){
            encontrado = true;
        }
        else{i++;}
    }
    SpeechFigure sf = ((SpeechEditPart) editP).getPrimaryShape();
    String path = sf.getFigureSpeechTextToFigure().getText();
    //-----
    IWorkbenchWindow dw = PlatformUI.getWorkbench().getActiveWorkbenchWindow();
    if (dw != null) {
        IWorkbenchPage page = dw.getActivePage();
        if (page != null) {
            File fileToOpen = new File(path);
            if (fileToOpen.exists() && fileToOpen.isFile()) {
                IFileStore fileStore = EFS.getLocalFileSystem().
                    getStore(fileToOpen.toURI());
                try {
                    IDE.openEditorOnFileStore( page, fileStore );
                } catch ( PartInitException e ) {
                }
            }
        }
    }
}
```

- *RunPlugin*: el objetivo de este plug-in es ensamblar <e-Tutor> GE, junto con el sistema <e-Tutor> ya existente, que permite ejecutar los tutoriales. Para ello, debemos recorrer el tutorial que hemos creado e ir instanciando cada una de las clases como clases del marco de aplicación de <e-Tutor>.

Lo primero que haremos será importar el código propio de <e-Tutor> a nuestro proyecto. Para ello, abriremos el archivo MANIFES.MF del proyecto con extensión .diagram.

Nos situamos en la pestaña *Dependencias* y en el cuadro *Imported Packages* pinchamos en *add*. Localizamos el paquete

donde estén ubicadas las clases de <e-Tutor> y hacemos click en OK. Es necesario que este paquete se encuentre en un proyecto en el mismo workspace que estamos trabajando.



Una vez hecho esto, podemos comenzar a generar el tutorial. Creamos una nueva clase dentro del paquete Tutor.Tutor.diagram.part denominada RunTutor. Al igual que las de los plug-ins anteriores debe implementar la interfaz IObjectActionDelegate y extender a ActionDelegate. El método run, en este caso, busca en la lista de editParts la que se corresponde con el comienzo del tutorial. Una vez encontrada, llamará al método *generaTutorial* pasándole el objeto Comienzo y otro objeto representando al propio Tutorial.

```

@Override
public void run(IAction action) {
    // TODO Auto-generated method stub
    DiagramEditor editor = (DiagramEditor) PlatformUI.getWorkbench().
        getActiveWorkbenchWindow().getActivePage().getActiveEditor();
    List<AbstractEditPart> ep = editor.getDiagramEditPart().getChildren();
    boolean encontrado = false;
    int i = 0;
    EditPart editP = null;
    while(i<ep.size() && !encontrado){
        editP = (EditPart) ep.get(i);
        if (editP instanceof ComienzoEditPart){
            encontrado = true;
        }
        else{i++;}
    }
    ComienzoEditPart comienzo = (ComienzoEditPart) editP;
    TutorialEditPart tep = (TutorialEditPart) editP.getParent();
    DiagramImpl nit = (DiagramImpl) tep.getModel();
    TutorialImpl ti = (TutorialImpl) nit.getElement();
    NodeImpl nic = (NodeImpl) comienzo.getModel();
    ComienzoImpl ci = (ComienzoImpl) nic.getElement();
    try {
        generaTutorial(ci,ti);
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

```

El código de *generaTutorial* se encarga de procesar el objeto Tutorial que le llega como parámetro. Crearemos un objeto tutorial de <e-Tutor> ETTutorial (en adelante nos referiremos a los objetos de <e-Tutor> de manera normal, teniendo en cuenta que sus clases siempre comienzan por ET). Una vez creado el objeto procesaremos los atributos del tutorial y los asignaremos al ETTutorial. Una vez hecho esto, el método utiliza el objeto Comienzo para localizar el primer Speech del tutorial. Para terminar, queda configurar una última opción: debemos establecer el elemento inicial del tutorial, que se corresponde con el Speech que acabamos de obtener. Sin

embargo debemos tratar este Speech y hacerle corresponder con un elemento de <e-Tutor>.

El tratamiento de los elementos del tutorial se realiza de manera recursiva con el método *trataElemento*. Este método es el que se encarga de hacer la correspondencia de la que hablamos anteriormente, procesando cada uno de los elementos.

```
private void generaTutorial(ComienzoImpl ci,TutorialImpl ti) throws IOException {
    // TODO Auto-generated method stub
    ETTutorial tutorial = new ETTutorial();
    if(ti.getLabelEnd() != null && !(ti.getLabelEnd().equals("")))
        tutorial.setLabelEndButton(ti.getLabelEnd());
    if(ti.getTitulo() != null && !(ti.getTitulo().equals("")))
        tutorial.setTitle(ti.getTitulo());
    if(ti.getLogoLocation() != null && !(ti.getLogoLocation().equals("")) )
        tutorial.setLogoLocation(ti.getLogoLocation());
    if(ti.getBg() != null && !(ti.getBg().equals(""))){
        Color c = null;
        String color = ti.getBg();
        c = dameColor(color);
        tutorial.setBlackboardColor(c);
    }
    SpeechImpl speech = (SpeechImpl) ci.getSigComienzoSpeech();
    tutorial.setInitialTutorialElement((ETTutorialElement)trataElemento(tutorial,
        new HashMap<EObjectImpl,ETTutorialElement>(), speech));
    tutorial.run();
}
```

El funcionamiento de *trataElemento* es sencillo y bastante intuitivo. Le llega un objeto como parámetro y, tras comprobar a qué clase pertenece, (Speech, QuestionPoint...), procesa este elemento atendiendo a las características de las clases de <e-Tutor>.

En el caso de los Speech la correspondencia es con un objeto tipo ETText. Se configuran todos los atributos, se busca el elemento siguiente y con él se vuelve a llamar a *trataElemento*. Si el objeto que llega por parámetros es un QuestionPoint le haremos corresponder con un ETQuestionPoint. En este caso debemos crear un objeto de tipo Input asociado al ETQuestionPoint que servirá para que el alumno introduzca sus

respuestas. El Input es configurable y toma los valores de configuración de los atributos del QuestionPoint. A continuación, debemos recorrer cada una de las Answers del QuestionPoint, y para cada una de las Answers, establecer su respuesta asociada y recorrer todos sus Feedback. La correspondencia con <e-Tutor> es inmediata: Answer con ETAnswer y DefaultAnswer con ETDefaultAnswer.

Por último, en caso de ser un Fin existen dos posibilidades. La primera que Fin vaya seguido de un objeto Tutorial (tutorial anidado), y por tanto el elemento siguiente a tratar sería el primero del nuevo tutorial. En este caso se vuelve a sacar el primer Speech y continúa el proceso. La segunda, si no tiene un Tutorial anidado, el procesamiento habría acabado.

Una vez hecho esto hemos contemplado todas las posibilidades y no quedará más que ejecutar el tutorial.

Cabe destacar que el método *trataElemento* lleva como parámetro un HashMap, formado por los elementos del diagrama y su traducción a <e-Tutor>, cuya función es evitar referencias circulares entre los elementos del diagrama.

```

private ETutorialElement trataElemento(ETutorial tutorial,
    HashMap<EObjectImpl, ETutorialElement> traducidos, EObjectImpl e) throws IOException {
    // TODO Auto-generated method stub
    ETutorialElement te = traducidos.get(e);
    if(te != null)
        return te;
    if(e instanceof SpeechImpl){
        SpeechImpl speech = (SpeechImpl) e;
        ETText t = new ETText();
        File f = new File(speech.getTexto());
        BufferedReader br = new BufferedReader(new FileReader(f.getAbsolutePath()));
        String s = br.readLine();
        String texto = new String();
        while(s != null){
            texto = texto+s;
            s = br.readLine();
            if(s!=null)
                texto = texto+"\n";
        }
        t.setText(texto);
        t.setTutorial(tutorial);
        if(speech.getDelay() != 0)
            t.setDelay(speech.getDelay());
        else
            t.setDelay(1000);
        if(speech.getBg() != null && !(speech.getBg().equals(""))){
            String cBg = speech.getBg();
            Color bg = dameColor(cBg);
            t.setBackgroundColor(bg);
        }
        if(speech.getFg() != null && !(speech.getFg().equals(""))){
            String cFg = speech.getFg();
            Color fg = dameColor(cFg);
            t.setForegroundColor(fg);
        }
        if(speech.getScaleFont() != 0)
            t.setFontScaleFactor(speech.getScaleFont());
        if(speech.isIsBold())
            t.setFontStyle(Font.BOLD);
        if(speech.isIsItalic())
            t.setFontStyle(Font.ITALIC);
        traducidos.put(speech, t);
        EObjectImpl siguiente = (EObjectImpl) speech.getSigFin();
        if(siguiente == null)
            siguiente = (EObjectImpl) speech.getSiguienteQP();
        if(siguiente == null)
            siguiente = (EObjectImpl) speech.getSigSpeech();
        t.setNextElement(trataElemento(tutorial, traducidos, siguiente));
        return t;
    }
    else{
        if(e instanceof FinImpl){
            FinImpl fi = (FinImpl) e;
            if(fi.getSigTut() == null){
                return null;
            }
            else{
                Tutorial tut2 = fi.getSigTut();
                Comienzo com = tut2.getTut_tiene_Comienzo();
                Speech speech = com.getSigComienzoSpeech();
                ETutorialElement elem = trataElemento(tutorial, traducidos, (EObjectImpl) speech);
                return elem;
            }
        }
        else{
            if(e instanceof QuestionPointImpl){
                QuestionPointImpl nodoqp = (QuestionPointImpl) e;
                ETQuestionPoint tqp = new ETQuestionPoint();
                traducidos.put(nodoqp, tqp);
            }
        }
    }
}

```

```

ETSimpleInput input = new ETSimpleInput();
if(nodoqp.getInputBColor() != null && !(nodoqp.getInputBColor().equals(""))){
    String cBg = nodoqp.getInputBColor();
    input.setBackgroundColor(cBg);
}
if(nodoqp.getInputFColor() != null && !(nodoqp.getInputFColor().equals(""))){
    String cFg = nodoqp.getInputFColor();
    input.setForegroundColor(cFg);
}
if(nodoqp.getInputLNB() != null && !(nodoqp.getInputLNB().equals(""))){
    String lnb = nodoqp.getInputLNB();
    input.setLabelNextButton(lnb);
}
if(nodoqp.getInputSFactor() != null && !(nodoqp.getInputSFactor().equals(""))){
    String sfactor = nodoqp.getInputSFactor();
    input.setFontScaleFactor(sfactor);
}
if(nodoqp.isInputIsBold())
    input.setFontIsBold("yes");
if(nodoqp.isInputIsItalic())
    input.setFontIsItalic("yes");
tqp.setInput(input);
EList<AbstractAnswer> ans = nodoqp.getMisRespuestas();
int numResp = ans.size();
tqp.setNumberOfAnswers(numResp);
for(int i =0;i<numResp;i++){
    ETAbstractAnswer taa = null;
    AbstractAnswer a;
    a = ans.get(i);
    if(a instanceof DefaultAnswer){
        ETDefaultAnswer tda = new ETDefaultAnswer();
        taa = tda;
    }
    else{
        if(a instanceof Answer){
            ETAnswer tans = new ETAnswer();

            String resp = a.getDato();
            tans.setResponse(resp);
            taa = tans;
        }
    }
    taa.setTutorial(tutorial);
    tqp.setAnswer(i, taa);
    EList<Feedback> feedb = a.getMisFeedback();
    int numFeedback = feedb.size();
    taa.setNumberOfFeedbacks(numFeedback);
    for(int j = 0;j<numFeedback;j++){
        Feedback fb = feedb.get(j);
        SpeechImpl sig = (SpeechImpl) fb.getMiSpeechDestino();
        String nR = fb.getNumRespuesta();
        int numF = Integer.decode(nR);
        numF = numF-1;
        te = trataElemento(tutorial,traducidos,sig);
        taa.setFeedback(numF, te);
    }
}
return tqp;
}
return null;
}
}
}

```

Para el correcto funcionamiento del generador tuvimos que añadir código al generado de manera automática por GMF. La razón es la siguiente: como se explicó en puntos anteriores, el mapeo en el modelo Gmfmap de la conexión Feedback es especial y no quedaba almacenada correctamente dentro del modelo. Tuvimos que incorporar código al EditPart del Feedback para que realizara esta función. En el paquete Tutor.Tutor.diagram.edit.parts dentro de la clase FeedbackEditPart debemos modificar el método *createConnectionFigure*.

```
protected Connection createConnectionFigure() {  
    FeedbackFigure f = new FeedbackFigure();  
    ConnectorImpl ni = (ConnectorImpl) this.getModel();  
    FeedbackImpl fi = (FeedbackImpl) ni.getElement();  
    AbstractAnswer a = fi.getMiAnswerOrigen();  
    EList<Feedback> mf = a.getMisFeedback();  
    if(!mf.contains(fi))  
        mf.add(fi);  
    return f;  
}
```

4.6.- Reconstrucción de <e-Tutor> GE

La versión de <e-Tutor> GE realizada durante este proyecto, no es una versión definitiva ni completa de la herramienta. Es una aplicación susceptible de ser ampliada o mejorada, introduciendo modificaciones que la hagan más atractiva y manejable para el usuario, o que amplíen la potencia de creación de tutoriales. Por este motivo se incluye esta sección en la memoria, orientando así a los futuros diseñadores de <e-Tutor> GE, en una próxima versión de la herramienta.

A la hora de rediseñar <e-Tutor> GE, los cambios fundamentales deben realizarse a nivel del metamodelo. Es aquí donde realizaremos modificaciones en la estructura de la aplicación, añadiendo nuevos elementos, nuevos datos a almacenar, etc.

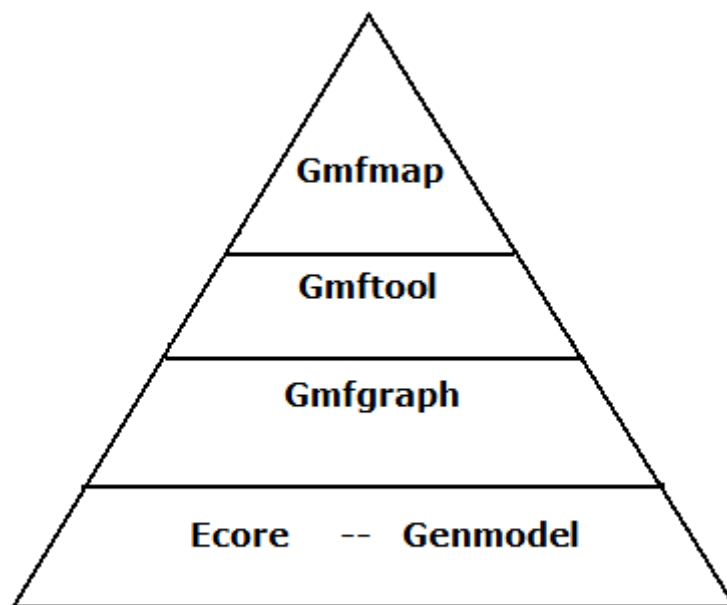


Figura 4.10 Jerarquía de modelos en GMF

La Figura 4.10 nos muestra una jerarquía de modelos en GMF. Con esto, queremos hacer ver que cualquier cambio que realicemos en un modelo implicará modificaciones en modelos de niveles superiores. Por ejemplo, un cambio en el modelo de gráficos (Gmfgraph), requerirá reconstruir tanto la paleta (Gmftool) como en el mapeo (Gmfmap). En nuestro caso, estamos hablando de modificaciones en el metamodelo (Ecore), por lo que cualquier cambio a este nivel requerirá una reconstrucción completa de la herramienta. Para ello debemos seguir los puntos 4.3, 4.4, y 4.5 de la memoria.

Comenzaremos por redefinir el metamodelo Ecore (4.3). Una vez lo tengamos, procederemos a reconstruir <e-Tutor> GE (4.4). Definiremos cada uno de los modelos (Genmodel, Gmfggraph, Gmftool, Gmfmap, Gmfgen), como se indica en la sección 4.4. Hecho esto pasaremos a generar el código de manera automática.

Para terminar, no quedará más que añadir los tres plug-ins a nuestra herramienta y las partes de código necesarias para su correcto funcionamiento. Este proceso viene detallado en la sección 4.5. Crearemos los tres proyectos de plug-in, las clases asociadas a ellos, y añadiremos los fragmentos de código necesarios en las clases indicadas.

Una vez hecho todo esto, <e-Tutor> GE quedará completamente reconstruido y podremos pasar nuevamente a su ejecución.

4.7.- Control de versiones de <e-Tutor> GE

El desarrollo de la herramienta está basado en un proceso iterativo, de manera que partiendo de una versión de <e-Tutor> GE básica se fueron incorporando nuevas extensiones. El objetivo de esta sección es reflejar cada una de estas versiones y las incorporaciones que se fueron haciendo a la herramienta básica.

4.7.1.- Versión básica <e-Tutor> GE

Esta versión de la herramienta incluía las funcionalidades más primarias para la creación del tutorial. Únicamente disponía de figuras cuadradas y de color blanco. Carecía de la conexión Feedback, no contaba con tutoriales anidados, las Answer eran figuras independientes (es decir, no se localizaban dentro de los

QuestionPoint), los Speech no tenían un archivo asociado, sino que su texto se introducía en un label, y por supuesto, tampoco contaban con la funcionalidad que aportan los tres plug-ins explicados anteriormente. Esta primera versión fue una aproximación al problema, pero sin embargo, fue una de las más costosas de conseguir, puesto que desconocíamos totalmente las tecnologías de GEF, EMF, y GMF. Constituyó la rampa de entrada al desarrollo de la herramienta.

4.7.2.- Modificación de figuras

El siguiente paso que seguimos fue introducir nuevas figuras, con diferentes colores, puesto que el manejo e interpretación de un diagrama con figuras todas iguales entre sí se hacía realmente complicada. Para ello incorporamos nuevas figuras como elipses y rectángulos redondeados, y establecimos un color concreto para cada una de ellas que perduró durante todo el desarrollo futuro.

4.7.3.- Incorporación de la clase Feedback

A pesar de tener un modelo básico, nuestro diagrama seguía sin estar completo, puesto que faltaba una de las piezas más importantes: la conexión Feedback. Para ello tuvimos que volver a nuestro metamodelo e introducir una nueva clase, ya que un Feedback debe guardar información concreta e introduciendo una simple conexión carecíamos de esta posibilidad. En este punto apareció otra de las dificultades importantes en el desarrollo de <e-Tutor> GE: debíamos mapear una clase como una conexión. Investigando dentro del modelo Gmfmap encontramos la solución y terminamos esta nueva

versión satisfactoriamente. Ahora ya podíamos crear tutoriales completos.

4.7.4.- Compartments y tutoriales anidados

Una vez que teníamos un editor funcional y completo, nuestro trabajo se centró en incorporar nuevas modificaciones y funcionalidades que lo hicieran más cómodo y atractivo para el usuario. Además, cambiamos la notación gráfica para hacerla más similar al modelo de <e-Tutor> que nos había sido proporcionado. Para ello introdujimos los compartments.

Los compartments son un tipo de figuras especiales que se utilizan para el almacenamiento de otras figuras. Así, un compartment es una especie de contenedor para otro tipo de figuras. Establecimos un compartment dentro de los QuestionPoint de manera que las Answers pudieran ir colocadas en el interior de ellos.

Por otro lado, introdujimos la posibilidad de anidar tutoriales. Para ello definimos un nuevo elemento Tutorial, que podía introducirse en nuestro diagrama de manera que al hacer doble click sobre él se abría una nueva ventana de edición para un nuevo tutorial. Así podíamos enganchar un tutorial con otro de manera cómoda e intuitiva.

4.7.5.- Speech con archivo y editor de texto

Seguimos añadiéndole funcionalidades a nuestra herramienta. Nos dimos cuenta de que con un simple label, el texto y la información que podían guardar los Speech era bastante reducida. Tomamos la decisión de que a los Speech se les pudiera asociar un archivo de texto, de manera se pudiera contener todo tipo de información en él,

desde un texto hasta código de programación que posteriormente pudiera ser procesado. El proceso que seguimos fue el siguiente:

- Primero asociamos un archivo por defecto al Speech de manera que en cuanto el usuario creara uno, ya estuviera relacionado con ese archivo.
- Posteriormente añadimos una acción sobre el objeto Speech, de manera que al hacer doble click sobre uno de ellos se abría una ventana de edición de texto referenciando al archivo que tuviera asociado. En esa ventana de edición de texto modificábamos el contenido del archivo y posteriormente lo guardábamos.

Sin embargo, este método trajo algunas complicaciones posteriores y en futuras versiones lo modificamos.

4.7.6.- Plug-in para cargar archivos

Pensamos que tener un archivo asociado por defecto al Speech limitaba mucho las posibilidades, por lo que introdujimos la posibilidad de cargar un archivo cualquiera. Para ello, creamos un plug-in, de manera que, haciendo click derecho sobre el Speech, dentro del menú contextual, aparecía una opción para cargar un archivo. Seleccionando esta opción se nos abría un cuadro de diálogo con un selector de archivos. Buscábamos el archivo deseado y lo abríamos. De esta forma el archivo quedaba asociado al Speech en lugar del anterior que tuviera. Así, podíamos crear archivos de manera independiente a la herramienta y luego irlos asociando a medida que creábamos el diagrama.

4.7.7.- Plug-in para editar archivos

Una vez creado el plug-in para cargar un archivo, creímos conveniente extraer la función para editar el archivo, de la que hablamos en el punto 4.6.6, e incorporarla en forma de un nuevo plug-in. De este modo, al hacer click derecho sobre un Speech, se abría el menú contextual con dos opciones: una para cargar un archivo al Speech, y otra para editar el archivo que tuviese asociado en ese momento. Al seleccionar la opción de editar se abría la ventana de edición de texto de la que hablamos anteriormente.

4.7.8.- Modificación del metamodelo para guardar los archivos

En un determinado momento, nos dimos cuenta de que había algo que no funcionaba. La información relativa al archivo del Speech debía quedar guardada en el modelo, cosa que no estábamos haciendo. Introdujimos cambios en el metamodelo y añadimos atributos, de manera que en el Speech quedaba almacenada la ruta del archivo que tenía asociado. Además esta información quedaba guardada en el modelo, por lo que era persistente y no se perdía a pesar de realizar varias ejecuciones.

4.7.9.- Plug-in de ejecución y modelo definitivo

Como último paso quedaba enganchar nuestra herramienta con <e-Tutor>. Para ello, debíamos recorrer nuestro diagrama e ir instanciando nuestros objetos como objetos de <e-Tutor>. Sin embargo, había ciertas diferencias entre ambos. Por lo que tuvimos que hacer algunas modificaciones en nuestro metamodelo:

-
- Introdujimos una nueva clase llamada DefaultAnswer, la cual utilizamos para referirnos a cualquier otra respuesta que diese el alumno, no explícitamente recogida dentro de las otras respuestas (Answers). También añadimos otra clase llamada AbstractAnswer de forma que tanto el Answer normal como el DefaultAnswer heredaban de ella.
 - Añadimos atributos de configuración. Esto se debe a que tanto los Speech, como los QuestionPoint y el Tutorial, tienen opciones configurables por el usuario, que pueden ser colores, tamaños de letra, etc.

Una vez hechas estas últimas modificaciones sólo quedaba enlazar el diagrama con <e-Tutor>. La manera de hacerlo fue con un plug-in. Al hacer click derecho en una parte en blanco del diagrama, aparece, en el menú contextual, una opción para ejecutar el tutorial. Esta opción recorre todo el diagrama y va instanciando los objetos en objetos de <e-Tutor>, como comentamos anteriormente. Una vez terminado el recorrido se ejecuta el tutorial que hemos creado.

5.- Conclusiones y trabajo futuro

Puesto que en apartados anteriores hemos explicado en qué consiste el proyecto, y también hemos comentado sus funcionalidades básicas, en esta sección vamos a establecer las principales conclusiones y el posible trabajo futuro.

5.1.- Conclusiones

El proyecto cumple todos los objetivos propuestos en el primer capítulo de la memoria, por tanto el resultado final es muy satisfactorio.

El desarrollo de la aplicación ha sido costoso, debido principalmente a una pronunciada pendiente de entrada al proyecto. A pesar de ello, se ha obtenido una herramienta de fácil manejo para el usuario y completamente funcional.

Además, se ha conseguido una aplicación modular, de manera que cualquier modificación futura que se quiera añadir no afecte significativamente al trabajo ya realizado.

A lo largo del desarrollo del proyecto hemos ido encontrando diferentes problemas que se han ido solucionando satisfactoriamente. El primero y principal fue el absoluto desconocimiento sobre Eclipse y sus tecnologías, tales como EMF, GEF y GMF. Tuvimos que dedicar mucho tiempo a su estudio y aprendizaje.

Resultó un tanto repetitivo el proceso de modificación de la herramienta, puesto que al añadir cualquier funcionalidad, debíamos repetir cada uno de los pasos para la creación de la aplicación. A medida que avanzamos descubrimos que los diferentes módulos

podían ser reutilizables, lo que supuso un ahorro considerable en tiempo.

A pesar de la cantidad de código que GMF genera de manera automática, tuvimos que añadir ciertos fragmentos de código para completar todas las funcionalidades que necesitaba <e-Tutor> GE. Además, esto requería de un estudio previo del código generado por GMF para comprenderlo y así poder realizar las modificaciones oportunas.

Por tanto, podríamos concluir diciendo que este trabajo nos ha permitido profundizar en las diferentes tecnologías de Eclipse, ampliándose nuestros conocimientos sobre esta plataforma. Gracias a esto, podemos afirmar que los tres frameworks utilizados conjuntamente suponen una potente herramienta para el desarrollo de modelos y entornos gráficos asociados a ellos.

Por último, decir que la experiencia personal adquirida durante la realización del proyecto ha sido especialmente enriquecedora. Resulta gratificante comprobar que los objetivos iniciales propuestos hace meses se han cumplido con éxito.

5.2.- Trabajo futuro

Hemos desarrollado una primera versión de <e-Tutor> GE que proporciona al usuario una serie de funcionalidades básicas. Sin embargo, esta primera versión es susceptible de ser ampliada para aprovechar todas las posibilidades que <e-Tutor> nos ofrece, como por ejemplo incluir imágenes y vídeos en el tutorial.

Además, el editor gráfico que se ha generado admite gran cantidad de mejoras, como una paleta más sencilla y cómoda para el usuario,

un aspecto gráfico más atractivo o un modo de ejecución más intuitivo.

Por último, sería interesante recortar las funcionalidades de Eclipse una vez ejecutada la aplicación. Actualmente, al ejecutar la herramienta se abre un entorno de desarrollo que incluye todas las posibilidades que Eclipse nos ofrece (creación de todo tipo de proyectos, multitud de vistas y de menús...). Sin embargo, sería más cómodo para el usuario un entorno de trabajo mucho más limitado, de manera que sólo se pudieran crear proyectos de <e-Tutor> GE, y en el que aparecieran una serie de menús, vistas y botones adaptados a nuestro proyecto.

6.- Referencias bibliográficas

- [1] Bork, A.: Personal Computers for Education. Harper & Rows. 1985
- [2] Clayberg, E.,Rubel, D.: Eclipse Plug-ins, Third Edition. Addison-Wesley Professional. 2008.
- [3] Cwalina, K., Abrams, B.: Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, Second Edition. Addison-Wesley Professional. 2008.
- [4] Daum, B.: Profesional Eclipse 3 para desarrolladores Java. ANAYA. 2004
- [5] Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Professional. 2009.
- [6] Ibrahim, B.: Software Engineering Techniques for CAL. Education & Computers 5, 215-222. 1989
- [7] Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley. 2008
- [8] Moore, B., Dean, D., Gerber, A., Wagenknecht, G., Vanderheyden, P.: IBM RedBook: Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework.
- [9] Piaget, J.: Biologie et connaissance. París: Gallimard. 1967
- [10] Sarasa, A., Sierra, J.L., Fernández-Valmayor, A.: Procesamiento de Documentos XML Dirigido por Lenguajes en Entornos de e-Learning. IEEE-RITA 4(3), 175-183. 2009
- [11] Schwaber, K., Beedle, M.: Agile Software Development with SCRUM, 1st Edition. Prentice-Hall. 2001
- [12] Sierra, J.L., Fernández-Manjón, B., Fernández-Valmayor, A.: A Language-Driven Approach for the Design of Interactive Applications. Interacting with Computers 20(1), 112-127. 2008
- [13] Sierra, J.L., Fernández-Valmayor, A., Fernández-Manjón, B.: From Documents to Applications Using Markup Languages. IEEE Software 25(2), 68-76. 2008
- [14] Sierra, J.L., Fernández-Valmayor, A., Fernández-Manjón, B.: How to Prototype an Educational Modeling Language. SIIE'07, 14-16 Nov., Porto, Portugal. 2007
- [15] Sleeman, D., Brown, J.S (eds.). : Intelligent Tutoring Systems. Academic Press. 1986
- [16] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling framework. Addison-Wesley Professional. 2008.
- [17] Vesperman, J.: Essential CVS. O'Reilly Media, Inc. 2003

7.- Webgrafía

A continuación listamos algunos sitios web que hemos consultado durante el desarrollo de este proyecto.

1. Sistema tutorial para el estudio de la fisiología del aparato respiratorio del cuerpo humano:
scielo.sld.cu/scielo.php?pid=S0864-21412004000300004&script=sci_arttext (último acceso 22 Junio)
2. Contenidos en e-learning: El rey sin corona:
www.gestiondelconocimiento.com/leer.php?colaborador=javitomar&id=246 (último acceso 22 Junio)
3. Software Engineering Techniques for CAL:
vlib.org/cuisung/CBL.papers/EdComp1/EdComp1.compact (último acceso 22 Junio)